

3 D O M 2 R E L E A S E ♦ V E R S I O N 2 . 7

Release 2.7

- ♦ *3DO M2 Release 2.7 Release Notes*
- ♦ *3DO M2 Mercury Programmer's Guide*
- ♦ *3DO M2 Command List Toolkit*
- ♦ *3DO M2 Link/Dump Programmer's Guide*



3DO M2 Release 2.7 Release Notes

Version 2.7 – June 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

3DO and the 3DO logos are trademarks and/or registered trademarks of The 3DO Company. ©1996 The 3DO Company. All rights reserved. Other brand or product names are the trademarks or registered trademarks of their respective holders.

Table of Contents

Preface	RLN-vii
About This Document.....	RLN-vii
About This Release.....	RLN-vii
Audience.....	RLN-viii
Organization of This Document	RLN-viii
Typographical Conventions	RLN-ix
1.0 Introduction to 3DO M2 Release 2.7	RLN-1
1.1 READ THIS NOW!	RLN-1
1.2 Major Features Added or Changed	RLN-1
2.0 Installing 3DO M2 Release 2.7	RLN-2
2.1 Notes	RLN-2
2.1.1 Post-installation Setup - Before Running the Debugger	RLN-2
3.0 Hardware.....	RLN-2
3.1 Caveats and Known Bugs	RLN-2
4.0 C Development Tools.....	RLN-3
4.1 Compiler	RLN-3
4.1.1 Caveats and Known Bugs	RLN-3
4.2 ANSI C Support	RLN-3
4.2.1 READ THIS NOW!	RLN-3
4.2.2 Caveats and Known Bugs	RLN-3
4.3 Link3DO	RLN-4
4.3.1 Caveats and Known Bugs	RLN-4
4.4 3DO Debug	RLN-4
4.4.1 Caveats and Known Bugs	RLN-4
4.4.2 Notes	RLN-5
4.4.2.1 Running the debugger in ROM mode	RLN-5
4.5 Comm3DO Application	RLN-6
4.5.1 READ THIS NOW!	RLN-6
4.5.2 Caveats and Known Bugs	RLN-6
4.6 CreateM2Make	RLN-6
4.6.1 READ THIS NOW!	RLN-6
4.6.2 Caveats and Known Bugs	RLN-6
5.0 OS	RLN-6
5.1 Kernel	RLN-6
5.1.1 Caveats and Known Bugs	RLN-6
6.0 Graphics.....	RLN-6
6.1 READ THIS NOW!	RLN-6
6.2 Caveats and Known Bugs	RLN-7
6.3 Mercury	RLN-8
6.3.1 READ THIS NOW!	RLN-8
6.3.2 Mercury Run-Time	RLN-9
6.3.2.1 Caveats and Known Bugs	RLN-9
6.3.2.2 Fixed Bugs	RLN-9
6.3.3 Mercury Examples	RLN-9
6.3.3.1 Caveats and Known Bugs	RLN-9

6.3.3.2	Fixed Bugs	RLN-9
6.3.4	Mercury Texture File Formats	RLN-9
6.3.5	Mercury Texture Tools	RLN-10
6.3.5.1	Caveats and Known Bugs	RLN-12
6.4	Graphics Folio	RLN-13
6.4.1	Features Added or Changed	RLN-13
6.4.2	Caveats and Known Bugs	RLN-14
6.5	Graphics State (GState)	RLN-14
6.5.1	Caveats and Known Bugs	RLN-14
6.6	Font Folio	RLN-14
6.7	3D Graphics Libraries	RLN-15
6.7.1	Framework	RLN-15
6.7.1.1	Caveats and Known Bugs	RLN-15
6.7.2	Pipeline	RLN-16
6.7.2.1	READ THIS NOW!	RLN-16
6.7.2.2	Caveats and Known Bugs	RLN-16
6.8	Command List Toolkit (CLT)	RLN-16
6.8.1	Caveats and Known Bugs	RLN-16
6.9	Geometry Compiler	RLN-16
6.9.1	Caveats and Known Bugs	RLN-16
6.10	Modeling Package Converters	RLN-16
6.10.1	Strata (ssptosdf)	RLN-16
6.10.1.1	Caveats and Known Bugs	RLN-16
6.11	PostPro	RLN-17
6.11.1	READ THIS NOW!	RLN-17
6.11.2	Caveats and Known Bugs	RLN-17
7.0	Video and Data Streaming	RLN-17
7.1	READ THIS NOW!	RLN-17
7.2	Data Streaming Run-Time Libraries	RLN-17
7.2.1	Data Stream run-time library	RLN-17
7.2.1.1	READ THIS NOW!	RLN-17
7.2.1.2	Caveats and Known Bugs	RLN-17
7.2.2	Subscriber library	RLN-17
7.2.2.1	SAudio Subscriber	RLN-17
	Caveats and Known Bugs	RLN-17
7.2.2.2	MPEG Audio Subscriber	RLN-18
	READ THIS NOW!	RLN-18
7.2.2.3	EZFlx Subscriber	RLN-18
	READ THIS NOW!	RLN-18
	Caveats and Known Bugs	RLN-18
7.3	Data Streaming Examples	RLN-18
7.3.1	PlaySA	RLN-18
7.3.1.1	Caveats and Known Bugs	RLN-18
7.3.2	VideoPlayer	RLN-18
7.3.2.1	Caveats and Known Bugs	RLN-18
7.3.3	EZFlxPlayer	RLN-19

7.3.3.1	Caveats and Known Bugs	RLN-19
7.4	Data Streaming Tools	RLN-19
7.4.1	AudioChunkifier	RLN-19
7.4.1.1	READ THIS NOW!	RLN-19
7.4.1.2	Caveats and Known Bugs	RLN-19
7.4.2	MPEGVideoChunkifier	RLN-19
7.4.2.1	Caveats and Known Bugs	RLN-19
7.4.3	MPEGAudioChunkifier	RLN-20
7.4.3.1	Caveats and Known Bugs	RLN-20
7.4.4	EZFlixChunkifier	RLN-20
7.4.4.1	READ THIS NOW!	RLN-20
7.4.4.2	Caveats and Known Bugs	RLN-20
7.4.5	QTVideoChunkifier	RLN-20
7.4.5.1	Caveats and Known Bugs	RLN-20
7.4.6	DATAChunkify -- the DATASubscriber's Data Prep Tool	RLN-21
Caveats and Known Bugs	RLN-21	
7.4.7	Weaver	RLN-21
7.4.7.1	Caveats and Known Bugs	RLN-21
7.4.8	Worksheet and Exercises	RLN-22
7.4.8.1	READ THIS NOW!	RLN-22
7.5	Video Tools	RLN-22
7.5.1	3-2 Pulldown, MovieEdit, MovieCompress	RLN-22
7.5.1.1	Caveats and Known Bugs	RLN-22
7.6	MPEGVideo	RLN-22
7.6.1	Features Added or Changed	RLN-22
7.6.2	Bugs Fixed	RLN-22
7.7	MPEG Audio	RLN-23
7.7.1	Bugs Fixed	RLN-23
8.0	Audio	RLN-23
8.1	Caveats and Known Bugs	RLN-23
8.2	Beep Folio	RLN-23
8.2.1	Caveats and Known Bugs	RLN-23
8.3	Audio folio	RLN-23
8.3.1	Caveats and Known Bugs	RLN-23
8.4	Music Library	RLN-23
8.4.1	Sound spooler	RLN-23
8.4.1.1	Caveats and Known Bugs	RLN-23
Appendix A	Examples in Tools Folder	RLN-25
A.1	Audio Examples	RLN-25
A.1.1	test3sf	RLN-25
A.2	Development Code Examples	RLN-25
A.2.1	Comm3DOExampleClient 68K	RLN-25
A.2.2	Comm3DOExampleClient PPC	RLN-25
A.2.3	C3 Task	RLN-25
A.3	Graphics Examples	RLN-25
A.3.1	M2 Kanji FontViewer	RLN-25

Appendix B Examples in Examples Folder	RLN-26
B.1 READ THIS NOW!	RLN-26
B.2 Audio Examples	RLN-26
B.2.1 Beep (in Audio:Beep)	RLN-26
B.2.2 Juggler (in Audio:Juggler)	RLN-26
B.2.3 MarkovMusic (in Audio:MarkovMusic)	RLN-26
B.2.4 Misc (in Audio:Misc)	RLN-26
B.2.5 Score (in Audio:Score)	RLN-27
B.2.6 Sound3D (in Audio:Sound3D)	RLN-27
B.2.7 SoundEffects (in Audio:SoundEffects)	RLN-28
B.2.8 Spooling (in Audio:Spooling)	RLN-28
B.3 Event Broker Examples	RLN-28
B.4 FileSystem Examples	RLN-29
B.5 Graphics Examples	RLN-29
B.5.1 CLT (in Graphics:CLT)	RLN-29
B.5.2 Fonts	RLN-29
8.4.2.1 Fixed Bugs	RLN-29
8.4.2.2 Fonts Examples (in Graphics:Fonts)	RLN-30
B.5.3 Frame (in Graphics:Frame)	RLN-30
B.5.4 Frame2d (in Graphics:Frame2d)	RLN-30
B.5.5 GraphicsFolio (in Graphics:GraphicsFolio)	RLN-31
B.6 Kernel Examples	RLN-31
B.7 Miscellaneous Examples	RLN-32
B.8 MPEG Examples	RLN-32
B.9 Data Streaming Examples	RLN-33

Preface

About This Document

This document describes technical changes, bug fixes, and known problems in the 2.7 release of the 3DO M2 tools and operating system. This document should be used in conjunction with the *3DO Release 2.0 Release Notes*.

The *Release 2.7 Release Notes* contains the following information:

- All warnings, caveats, and known bugs that apply to the product at the Release 2.7 level. This includes warnings, caveats, and known bugs that were documented for Release 2.0 and are still applicable in Release 2.7.
- Features that were added or changed since Release 2.0.
- Bugs that were fixed since Release 2.0.
- Notes specifically for Release 2.7.

The *Release 2.7 Release Notes* DO NOT CONTAIN the following information:

- Supplementary documentation and tutorial-type information that was included in the *Release 2.0 Release Notes*. See the *Release 2.0 Release Notes* for that information.
- Features that were added or changed or bugs that were fixed in Release 2.0. See the *Release 2.0 Release Notes* for that information.
- Notes relating to Release 2.0. See the *Release 2.0 Release Notes* for that information.

The *Release 2.7 Release Notes* appear in the form of a hardcopy document and also in the form of an ASCII document on the distribution media. The hardcopy version may be slightly more recent. Any differences between the hardcopy and electronic versions of this document are indicated by change bars in the hardcopy version.

In addition to this document, there are component-specific release notes for some components in the form of ReadMe files on the distribution media. Read this document and the ReadMe files carefully because they contain important caveats and warnings that can make the difference between your code running or failing.

Note: *The hardcopy version of this document is more recent than and takes precedence over the electronic version and the ReadMe files.*

About This Release

This 3DO M2 2.7 release is an update to the 2.0 release and provides a number of enhancements and bug fixes.

This release includes the following items:

- M2_2.7 Software CD
- *3DO M2 Release 2.7 Read This First* document
- *3DO M2 Release 2.7 Release Notes*
- Some additional and replacement manuals for the 3DO M2 Developer's Documentation Set that was issued with Release 2.0. The complete document set was not reissued for Release 2.7.

The new and replacement manuals provided with Release 2.7 are as follows:

- *3DO M2 Mercury Programmer's Guide*
- *3DO M2 Command List Toolkit*
- *3DO M2 Link/Dump Programmer's Guide*

Audience

This document is for developers preparing titles for the M2 Development System.

Organization of This Document

- **Preface**

Brief description of this document.

- **Introduction to 3DO M2 Release 2.7**

Notes about the release as a whole.

This section contains some or all of the following subsections:

- **Introductory paragraph(s).**
Brief overview or description of this release.
- **READ THIS NOW!**
Particularly urgent information about this release.
- **Major Features Added or Changed**
Particularly important features added or changed in this release.
- **Major Bugs fixed.**
Particularly important bugs fixed in this release.
- **Major Caveats and Known Bugs**
Particularly important known bugs or things to watch out for in this release.
- **Notes**
Additional information about this release.

- **Installing 3DO M2 Release 2.7**

Notes on installing this release. This section contains some or all of the following subsections concerning the installation process:

- Introductory paragraph(s).
- READ THIS NOW!
- Features Added or Changed
- Bugs fixed.
- Caveats and Known Bugs
- Notes
- **Sections on individual components of the 3DO M2 system** (e.g., Hardware, C Development Tools, Graphics, Video and Data Streaming, Audio).

Each section contains subsections about subcomponents. Each section or subsection contains some or all of the following subsections relating to the overall component or individual subcomponent:

- Introductory paragraph(s).
- READ THIS NOW!
- Features Added or Changed
- Bugs fixed.
- Caveats and Known Bugs
- Notes
- **Appendices**
Additional information.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>Scene_GetStatic(scene)</code>
procedure name	<code>Char_TotalTransform()</code>
new term or emphasis	In M2, <i>characters</i> are objects that can be displayed on the screen.
file or folder name	It is located in "folder:subfolder".

1.0 Introduction to 3DO M2 Release 2.7

Release 2.7 of the 3DO M2 Portfolio and Toolkit consists of new, updated, and improved software and documentation.

The following sections contain information about the release as a whole. Information about individual components of the M2 system are contained in subsequent sections.

1.1 READ THIS NOW!

- **CDs created using Release 2.0 will NOT run on systems running Release 2.7.**
- **M2 Release 2.7 runs on Rev G or later dev cards. It does not run on earlier dev cards.**
- The complete document set was NOT reissued for Release 2.7. See "About This Release" on page vii in the Preface of this document for a list of the new and replacement manuals that are included in M2 Release 2.7:
- Setting the Current Directory

The current directory of a program is now set by the system to refer to the directory where the executable being run is located. That is, the current directory of a program may be different than the current directory of the shell used to invoke the program.

This behavior is intended to make it easier to write programs that don't rely on knowledge of their location. All of the files for a program can be accessed relative to the program's initial current directory, since it is assured that the current directory will always point to where the application is.

This behavior can be defeated by specifying the `-Hflags=1` command-line argument when linking your programs. This causes the linker to set a bit in the executable that tells the system to initially set that program's current directory to be the same as that of the shell it was launched from. This behavior is what a normal shell environment typically does, and is desirable for any shell utility programs, such as "ls".

1.2 Major Features Added or Changed

- Mercury Rendering Engine

This release of M2 includes the Mercury Rendering Engine, a 3D graphics pipeline that is optimized to run in the 4K instruction cache of the PowerPC (PPC602) CPU that drives M2.

Although Mercury is designed to be used by programs written in C or C++, it runs at a speed approaching that of assembly language because all of its internal procedures are written in assembly language and pass arguments by means of registers.

Mercury is offered as an alternative to the original M2 Graphics Framework and Graphics Pipeline development packages, which are more traditional, more programmer-friendly but slower-running C-language APIs. Developers who want to create games that run at the fastest possible speed should consider using the Mercury API. Mercury is documented in the *3DO M2 Mercury Programmer's Guide*, which is also new with this release.

2.0 Installing 3DO M2 Release 2.7

This section describes changes to the installation process.

2.1 Notes

2.1.1 Post-installation Setup - Before Running the Debugger

To be able to run the debugger on an **8 Mb dev card**, you must do the following:

1. Copy the file "ROM.m2.flash.debug" into your dev card's Flash ROM. You must use the FlashROMTool to do this.
 - Run the **FlashROMTool**.
 - In the window titled "Onboard Flash", click in the Filename box.
 - Select the file "ROM.m2.flash.debug".
 - Click the Flash button.
 - When you see the "Verified" message, quit the FlashROMTool.You only need to do this once. You do not need to perform this first step again unless you receive a new OS release from 3DO or you use the FlashROMTool to copy another file into your dev card's flash ROM.
2. Set up the debugger.
 - Run the debugger. You must use debugger version 2.1a10 or later.
 - Under the Target menu, select Setup.
 - Make sure the ROM is set to "Debugger".
 - Click on the Script button and select the file "**debugger.flash.scr**".

To be able to run the debugger on an **16 Mb dev card**, you must do the following:

1. Set up the debugger.
 - Run the debugger. You must use debugger version 2.1a10 or later.
 - Under the Target menu, select "Setup".
 - Make sure the ROM is set to "Debugger".
 - Click on the Script button and select the file "**debugger.16M.scr**".

3.0 Hardware

3.1 Caveats and Known Bugs

- **Flashing Boot Screen**
When booting version G of the 3DO Development Card with 3DODebug or Comm3DO, the 3DO M2 boot screen will flash. This problem is related to the video output hardware, and is confined to the boot screen. It will cease once you launch an application and will not affect your work.
- The following bug is known to occur with all development cards and could cause visual artifacts noticeable to developers using this development system.
LOD determination can be incorrect for thin triangles.
In very thin (narrow) triangles, some spans will not have any pixels drawn on them, making the lines appear dashed. This is expected. The problem is that the vertical LOD logic is not ignoring these lines for the purposes of

generating VLodLatch and VLodCompare signals. So a VLodCompare signal is being generated to compare against a previous line's VLodLatch, except that previous line had no pixels in it.

The result is that the LOD determination made by the Texture Mapper based on horizontal and vertical u,v deltas may be incorrect for these triangles.

When a texture is displayed at extreme perspective (as on the side of a cube), the image exhibits blockiness in the vertical direction. This means that you see visible banding when looking at a vertical column of pixels, while a horizontal row of pixels looks fine.

The cause has to do with the LOD determination performed by the Triangle Engine. The triangle engine looks at the texel delta in both the horizontal and vertical direction, and chooses the coarser LOD. In the case described above, there is significant compression of the texture in the horizontal direction, but little compression in the vertical direction. This means that the LOD is determined by the horizontal delta, with the effect that, vertically, a coarser LOD is used than is desired. This means that several successive vertical pixels will sample the same texel in the coarser LOD, resulting in blockiness in the vertical direction.

The same effect may be seen at the top of the cube, if the top face is just barely visible, but this time the vertical compression drives the LOD choice, and the blockiness appears in the horizontal direction.

There is no way to avoid these problems, but generally when triangles are so thin as to cause this to occur, the texture will already be so warped (such as on the side of a barely visible cube) that the overall degradation of the image should be slight.

4.0 C Development Tools

4.1 Compiler

4.1.1 Caveats and Known Bugs

- If you abort a compilation using Command-., the temporary asm files in "{TempFolder}" are not deleted. You must manually delete these files to avoid filling up your disk.

Note that "{TempFolder}" is the folder "RAM Disk:temp:" if that folder exists. Otherwise, "{TempFolder}" is "{Boot}System Folder:Preferences:MPW:TempMPW:".

4.2 ANSI C Support

4.2.1 READ THIS NOW!

Strict ANSI checking does not allow C++ style comments `///
To allow such comments, do not specify the dcc flag -Xstrict-ansi.`

4.2.2 Caveats and Known Bugs

- Floating point exceptions are not ignored by default. If you are using floating point numbers, or routines that use floating point numbers (e.g., Graphics, or Audio routines), you should consider installing a floating point exception handler. The default action for floating point exceptions is to terminate the task, which may not be desirable.

- The routine “sqrtff” does not handle infinities very well. If your math calculations could involve an infinity, you may need to install an exception handler.

4.3 Link3DO

4.3.1 Caveats and Known Bugs

- Under certain circumstances, Link3DO relocates assembly language branches incorrectly.

The linker shipped on the M2_2.0 CD (version 1.15) relocates assembly language branches using truncated offset values under the following conditions:

- You are programming in assembly language -- e.g., building or using a custom version of the Mercury library
- You use a relative branch (such as bge or bge1) to an imported symbol which, at link time, is more than 32K bytes away from the branch.

Link3DO 1.15 will not detect the problem and will relocate the branch using a truncated offset value. This can cause a crash or other type of error because of a wild run-time branch.

This problem will only affect a small number of developers. If the problem affects you, try the new “experimental” linker, which fixes this problem. To do this,

- Locate your current (i.e., M2_2.0) Link3DO by typing “which Link3DO” in your MPW worksheet and pressing enter. Your Link3DO should be in your 3DO folder with a path such as
HardDisk:3DO:tools:M2_2.0:MPWTools:Link3DO.
- Rename this linker “Link3DO.orig” and drag the new (i.e., M2_2.7) Link3DO into the same folder.
- Before building your project again, quit and re-launch MPW.
- You should then do a clean (i.e., “full”) build of your project.

The new linker will report an out-of-range error, if one is present, and give you information that allows you to correct it.

4.4 3DO Debug

4.4.1 Caveats and Known Bugs

- On a Power Macintosh, selecting the menu item “File/Directories/Current Source” will cause a crash under some circumstances.

The circumstances leading to the crash are quite common (e.g., a “.spt” script has been run), and result from the default setup created when you use CreateM2Make. We recommend that you avoid selecting this menu item (File/Directories/Current Source).

- In the Target Setup dialog, the check box item “Initial bp in boot code” should never be selected. This functionality is broken, and will cause your target system to hang during boot.
- In the Source window, the PC arrow will only show up if it is at the first instruction of a source statement. Because of a bug in the line information, the PC arrow will occasionally disappear when source stepping. This most commonly occurs when stepping through “while” and “for” loops.

-
- Source files larger than 32K will not display correctly in the Source window.
 - The debugger has trouble switching between development cards. If you have two or more cards in your Mac, you can switch between cards by choosing "setup" from the "target" menu. A dialog will appear. The topmost pulldown menu is the "target." At this point, you can switch cards. Then click "OK" in the dialog and you will notice the message "M2 released from reset" in the Terminal window of the debugger. At this point, you must quit the debugger and re-launch it.
 - 3DODebug will report a syntax error when trying to display a variable name that is also the name of a field in a structure.

4.4.2 Notes

- Contrary to what users might expect, the debugger's "H/W Reset" command does not reset the Tasks list. This means that if you use the "debug" command to debug or run a program, you should execute Reset before the program quits. Otherwise, the debugger thinks the program is still running. If you then try to run that program without re-executing the "debug" command, the debugger acts as if you had used the "debug" command anyway and spends time loading symbols and setting the initial breakpoint.

This is not a bug; the debugger is behaving the way it is supposed to. What is at issue here is the name of the Task list window. Actually, the Task window displays a list of symbol files rather than an actual list of tasks.

4.4.2.1 Running the debugger in ROM mode

You may run your dev card in "ROM mode." In this mode, the dev card acts very much like a consumer unit. The debugger is not functional in ROM mode. You can, however, boot an M2 CD-ROM as it would boot on a real consumer unit.

To run in ROM mode:

- Run the FlashROMTool.
- In the window titled "Onboard Flash", click in the "Filename" box.
- Select the file "ROM.m2.unenc".
- Click the "Flash" button.
- When you see the "Verified" message, quit the FlashROMTool.
- Run the debugger. In the "Target" menu, select "Setup".
- Switch the "ROM" from "Debugger" to "Flash".
- Select the script "flash.scr".
- Click on "OK".
- The system will boot in ROM mode. You will not see any messages on the debugger console.

If you wish to switch back to debugger mode, you will need to use the FlashROMTool again to copy the file "ROM.m2.debug" into your dev card (see section 2.1.1 on page 2). Then in the debugger Target/Setup menu, switch the "ROM" back to "Debugger" and switch the script back to "debugger.flash.scr".

4.5 Comm3DO Application

4.5.1 READ THIS NOW!

Comm3DO App cannot be run simultaneously with 3DODebug.

4.5.2 Caveats and Known Bugs

- Certain makefiles, including Comm3DO when trying to compile the Comm3DO client task, have problems using Link3DO. The workaround is to remove the -m2 option from the link3do line in the makefile.

4.6 CreateM2Make

4.6.1 READ THIS NOW!

Developers are expected to create new makefiles with each release of the software CDs.

4.6.2 Caveats and Known Bugs

- Don't create any makefiles that request optimization and debugging to be turned on simultaneously.
- Some compiler options have been removed, because they are now stored in the config file.
- The makefile now creates a "<>.spt" file along with the executable. This script tells 3DO Debug where to find source code and symbols. Note that this does not work with multiple source directories. See the 3DO Debugger manual for more information.
- On a PowerPC Macintosh, attempting to view source directories if they are set using a ".spt" file will crash the Macintosh with an illegal instruction error. Note that CreateM2Make automatically generates a ".spt" file.

This is not a problem if you are using a 68k Macintosh or if you are viewing the directory when it is set manually.

5.0 OS

5.1 Kernel

5.1.1 Caveats and Known Bugs

- The debugging console view must currently always be opened full-screen due to a hardware bug.
- The operating system sometimes allocates memory in a way that causes fragmentation of the heap. This can create problems for applications that need large, contiguous memory allocations.

A workaround is to preallocate large, contiguous allocations.

6.0 Graphics

6.1 READ THIS NOW!

- When linking a graphics program, the following link line should be used:
`-lmusic -lframe2d -lframe_utils -lframework -lbsdf -lpbsdf -lframework
-lpipeline -lframe_utils -lbsdf -lpbsdf -lclt -lspmath -lfile
-leventbroker -lc`

-
- Minimizing Footprint: to decrease your memory footprint, keep in mind the following:

- use indexed textures whenever possible
- make sure that textures, materials and models are shared wherever possible.

- Interfacing with C++

The Graphics header files are designed for inclusion in C++ programs. In order to include the Graphics header files from C++ you should make sure that the preprocessor variable "GFX_C_Bind" is defined in your makefile.

6.2 Caveats and Known Bugs

- The instructions to run the "anim" program that are included in the header of the source file "anim.c" are out of date.

To run anim, do the following:

- Build the anim program using the default makefile or a makefile created by CreateM2Make.

If you are using a new makefile created by CreateM2Make, move your "anim" executable (and ".spt" file) into the following folder:

3DO:3DO_os:M2_2.7:Remote:Examples:Graphics:Frame:Anim

The skeleton.csf file is already in the directory by default

You can also leave the anim program where the CreateM2Make makefile placed it, and move or copy the skeleton.csf file needed to run the example into the same directory.

- Launch 3DODebug and navigate to the directory containing the anim executable.
- At the command line, type
anim skeleton.csf skeleton_world

The usage message that appears if you enter the command incorrectly is also out of date.

The correct usage is:

anim <filename>.csf <filename>_world

- The introduction to Chapter 5 of the *M2 Graphics Tools* manual contains a list of the graphics tools available in M2, along with a brief but useful explanation of what each tool is designed to do. Unfortunately, this useful and comprehensive list is difficult to find if you don't know exactly where to look for it because the manual's Table of Contents does not contain any reference to it.
- The footnote on Page 34 of the *M2 Graphics Tools* manual is incorrect and should be deleted. The M2 development hardware now supports compressed textures.
- In the *M2 Graphics Tools* manual Appendix D, on Page 255, the second typedef on the page (the typedef of the Seq_Action struct) and the warning note that follows should be changed to read as follows:

```
typedef struct {
    PackedID ChunkType; /* 'ACTN' */
    uint32 dataSize;
    float rate; /* Frames per second */
    uint16 numPOV;
    uint16 numExposures; /* Number of frames referenced below */
    uint16 frameRefNums[];
    Seq_Exp exposures[]; /* Optional */
} Seq_Action;
```

Warning: The following chunk is currently not supported. It is
provided for discussion and example purposes only.

6.3 Mercury

The texture lib, texture tools, gcomp program (called gmerc for Mercury), and converters have all been updated with new functionality and bug fixes to work with Mercury. The versions of these tools in the Mercury folder on the CD are more up to date than the versions in the Graphics folder.

6.3.1 READ THIS NOW!

- When using Mercury code, use the link3DO linker program contained in the Mercury Release folder.

See section 4.3.1 on page 4 of this document for a more detailed description of the problems that can be encountered by using the regular link3DO with Mercury code.

- At this time, the Mercury rendering engine has not been completely integrated with all of the older graphic components. In this release, conflicting definitions in some of the Mercury header files can cause problems when using fonts with the Mercury engine.

The conflicting definitions are "Color4" in "lighting.h" and "Point3" in "matrix.h".

You can avoid these problems by editing the two header files as follows:

- In "mercury/lighting.h" add

```
#ifdef MACINTOSH
#include "graphics:gp.h"
#else
#include "graphics/gp.h"
#endif
```

and remove the "Color4" struct definition.
- In "mercury/matrix.h" remove the following two lines:

```
#define Point3 Vector3D
#define pPoint3 pVector3D
```

6.3.2 Mercury Run-Time

6.3.2.1 Caveats and Known Bugs

- `Vector3D_GetX()`, `Vector3D_GetY()`, and `Vector3D_GetZ()` get a syntax error when compiling.

Workaround: Remove the incorrect ending semicolon from three "define" statements in "matrix.h":

Change this:

```
define Vector3D_GetX(v) (v)->x;  
define Vector3D_GetY(v) (v)->y;  
define Vector3D_GetZ(v) (v)->z;
```

to this:

```
define Vector3D_GetX(v) (v)->x  
define Vector3D_GetY(v) (v)->y  
define Vector3D_GetZ(v) (v)->z
```

- `Matrix_Zero()` causes a crash when used.

Workaround: Do not use this function. Instead, you can write your own code to zero out values in a matrix.

6.3.2.2 Fixed Bugs

- `Matrix_Mult()` now returns correct values.

6.3.3 Mercury Examples

6.3.3.1 Caveats and Known Bugs

- `helloworld` and `viewer` do not scale their input models. This makes the default model for these examples very small.
- In `helloworld` and `viewer`, setting the initial camera location to "0, 0, 0" causes a program exception. Avoid setting it to "0, 0, 0."
- `helloworld` and the `viewer` do not display models with transparency correctly because they do not initialize the source address field "srcaddr" of the current frame buffer. As a result, this field contains a random value.

Workaround: Edit the example code to initialize the "srcaddr" field. Consult the *3DO M2 Mercury Programmer's Guide* for a description of how to do this.

- `viewer` does not calculate bounding boxes correctly. This may cause erroneous warnings that you have a zero-size bounding box.

Workaround: Edit the example code, and insert your own code to calculate the bounding box.

6.3.3.2 Fixed Bugs

- Binary SDF models with a single pod no longer get a program exception when used with the `viewer` example.

6.3.4 Mercury Texture File Formats

As most M2 developers know, a UTF (unified texture format) file is an M2-specific file based on the IFF file format. It's an IFF format of the FORM type with the four-letter designator "TXTR".

Mercury users must also become familiar with the concept of a *UTF page file*. A UTF page file is simply a UTF texture file (that is, a FORM-type "TXTR" file) with an additional page chunk designated "M2PG". This chunk contains information about the sub-textures contained in the texel data chunk (now thought of as the texture page). M2PG is meant to be easy to parse and to be very general-purpose and not subject to change from release to release.

Additionally, there is the concept of a *Mercury page file*. This is a UTF page file with an additional chunk designated "PCLT". This chunk is very implementation-specific and will probably change as Mercury evolves. Think of this chunk as a compilation of all the data necessary to render a sub-texture within a texture page. This chunk is extremely hard to parse as it is mostly CLTs. Right now, the creation of this chunk is considered to be a one-way street and the M2 Texture Library and texture tools will merely skip over this chunk when they see it. When this chunk is present, the header chunk and TAB chunk are not needed and can be stripped out of the final texture. Future implementations of *utfpage* and the texture reader will eliminate the need of the DAB and M2PG chunks from the final textures as well, simplifying reading and processing of incoming textures.

IFF CAT files are simple concatenations of standard IFF FORMs. In this case, we are only interested in concatenations of "TXTR" FORMs, which are the UTF textures.

IFF LIST files are a more sophisticated way of storing multiple FORMs in a single file and allow for sharing of properties between FORMs. For example, an IFF LIST of UTF textures could contain twenty texture pages which share the same PIP. This would mean that only one copy of the PIP data ever need be stored in the file, thus saving the space required by nineteen PIP chunks. Because IFF LISTs are MUCH more complicated to parse (and may be too inefficient for a runtime environment) the current texture reader in Mercury doesn't support IFF LIST files. Adding IFF LIST support will be investigated and will become part of the runtime reader if its performance is satisfactory.

6.3.5 Mercury Texture Tools

This section describes several Mercury tools and explains how they handle various file formats.

- *utfpage*

This is the workhorse. It takes an ASCII texture file (*TexArray* and *TexPageArray*) as input and uses it to construct a Mercury page file. If multiple pages are specified, *utfpage* concatenates them into an IFF CAT file that can be read by the mercury reader.

Features:

Any Texture Application setting that is set in the *TexArray* or the texture itself is converted to a *CltsSnippet* and becomes part of the texture. This includes settings such as *WrapModes*, *PIPSelect*, *Blend Operations*, and the like. It should be noted that the settings in the *TexArray* have precedence over those in the texture file (which are stored in the TAB chunk). One upshot of this is that it is no longer necessary to manually set the *WrapMode* settings of a texture before creating the page. All that is necessary is to set proper *WrapMode* in the *TexArray* (and this field is set by all the 3D converters provided by the Tools group) entry for that texture.

One of the input options to the `utfpage` tool is an external PIP. Why would you want to specify a custom PIP for page processing? To allow for each sub-texture within a page to have its own unique PIP. Because the `TexArray` entry for a sub-texture can specify a PIP Index Offset, it's possible to have a multiple PIPs represented within the page's PIP. For example, four six-bit textures have at most 256 colors total between them. That means that all four textures PIPs could be represented entirely with a single PIP that is a concatenation of all the PIP. To create this concatenated PIP, one can use `utfpipcat`. Then that PIP file can be passed on to `utfpage`. The only other necessary step is to set each texture's `TexArray` entry to have the appropriate `PIPIndexOffset`. The first texture will have a `PIPIndexOffset` of 0 (so it doesn't need to be set), the second will have an offset of 64, the third 128, etc. Rather than create a PIP by hand and manually compute the `PipIndexOffset` for each `TexArray` entry, one can use the `-superpip` option. This option takes all the subtextures' PIPs in a page and concatenates them into a single PIP for the page. The restriction that the total number of colors of all the PIPs cannot exceed 256 is still in effect. This is a quick way to make shared PIPs, but the result may not be optimal since each texture in the page now has it's own exclusive PIP. In most applications, many textures are bound to have a common PIP and these values could be shared. The `-superpip` option, as handy as it is, makes no attempt to find these.

`utfpage` has a new option, `-superpip`. This option is extremely useful for creating pages quickly without having to worry about having a single PIP for the whole page. This option concatenates each page's sub-textures' PIPs into a single PIP for each page. It also automatically sets the `PIPIndexOffset` for each sub-texture so that each texture will reference it's own unique PIP. The restriction is that the number of colors of all the sub-textures' PIPs cannot exceed 256. For example, if you have a bunch of 64-color textures, you could have four per page with the `-superpip` option. The sub-textures need not share the same bit depth or number of colors so you could have two 64-color textures and still have room in the PIP for 8 16-color textures. You could even have two eight-bit textures as long as their total number of color didn't exceed 256 (one could have 100 colors and the other 156, for example).

Limitations:

All the textures in a given page must share the same PIP (one way or another) and the sum of their texel data cannot exceed 16KB. Right now, each page uses the first valid PIP of a sub-texture to be the PIP for the whole texture UNLESS a PIP is specified in the command-line options (see above) or the `-superpip` option is used. If a PIP is specified in the command-line options, that PIP will be used for each texture page.

- `utfunpage`

This recreates the original textures in a UTF page file, however, it currently only works on single page files. Texture page CAT files (*i.e.*, a file with multiple texture pages) must first be run through `utfsplit`. This will be changed shortly.

- `utfsplit`

This takes UTF files which are either in CATs or LISTs (*i.e.* multiple UTF textures in a single file) and splits them into individual UTF texture files.

- `utfquantmany`

Takes a series of images and find a common pip for them all and does color reduction if necessary. There is an option to quantize the input textures (and overwrite them) to that resulting PIP.

- `utfpipcat`

Concatenates the PIPs of several input files into a single PIP. Useful for have unique PIP among textures within a texture page. See below.

- `utfpipsub`

Replaces the ssb and alphas in a PIP with one that the user specifies. Useful if you want all entries in a PIP to have say SSB set to 1.

6.3.5.1 Caveats and Known Bugs

- `quantizer` bug:

The latest version of `quantizer` will crash upon exit on the Macintosh platform. `quantizer` will write out the proper quantized texture before crashing, and you can recover from the crash if Macsbug is installed.

To work around this problem, you can use `utquantmany` instead of `quantizer`. `utquantmany` acts like a multiple-file version of `quantizer` that also performs the duties of `utmakepip`.

This bug can be eliminated from `quantizer` by commenting out the last two `M2TX_Free` calls in the source code (immediately after the `M2TX_WriteFile` call) and recompiling the tool.

This bug is being fixed for a future release.

- `utffit` bug:

`utffit` can be made to crash by doing one of two things.

- Failing to provide in the argument list the number of LODs to generate.

If the number of LODs to generate is omitted (a common usage mistake), `utffit` evaluates the next argument (usually a string) as a number and comes up with 0. A division by 0 error will eventually occur.

- Giving `utffit` an impossible task.

If you give `utffit` an image that won't fit into 16KB, given the conditions specified by the user, the program will also get a division by zero error when it attempts to resize the image to have a 0 dimension.

For example, if `utffit` is asked to create an image that has 11 LODs, there is no way such an image can fit into 16K because, to have 11 LODs, the image must be at least 1024x1024 (2^{10}). `utffit` will try to downsize the image all the way down to 0x0 to create a 16K image, and this results in a division by 0.

This bug is being fixed for a future release.

- If you enter the command "`ppmtoutf`" with no parameters in order to get usage information on a Macintosh or an SGI machine, you will not get a usage prompt.

To stop the command, you must interrupt it, using Command-period on a Macintosh or Control-c or Delete on an SGI machine. Entering and interrupting the command as described here should not cause any problems for your system.

6.4 Graphics Folio

6.4.1 Features Added or Changed

This release completes the core functionality for the Graphics Folio. The new features in this release are as follows:

- Added non-interlaced Projector support. You can now create and display Views on non-interlaced displays.
Consequently, two new Projector types now exist:
 - NTSC-nolace
 - PAL-nolace
- Added the support code required to let applications to switch between Projectors on the fly. No special cooperation is required between Tasks; the folio handles everything.
- Added new functions that let applications use and manipulate Projectors:
 - `ActivateProjector()` and `DeactivateProjector()`
These functions turn Projectors on or off, making their View hierarchy visible or invisible.
 - `NextViewTypeInfo()`
Lets an application iterate through all the View types a Projector can support. You can use this to search the database intelligently for the View type most appropriate for the application.
 - `NextProjector()`
Lets an application iterate through all the available Projectors in the system.
 - `QueryGraphics()`
Makes available to applications some of the Graphics Folio's internal state information -- such as the current default Projector.
- You can now call `OpenItem()` and `CloseItem()` on a Projector. You have to do this if you wish to make a different Projector visible using `ActivateProjector()`.
- Added a new field to the Projector Item structure: `p_Flags`.
This field currently reports whether or not the given Projector is active.
- Increased the priority of the VBL interrupt to 210 because it has to occur very early to effect display mode changes.
- Added two new flags to the `vti_Flags` field in the `ViewTypeInfo` structure:
 - `VTIF_ABLE_AVG_H`
Set if the View can do horizontal pixel averaging.
 - `VTIF_ABLE_AVG_V`
Set if the View can do vertical pixel averaging.
- Added a new error code to the folio: `GFX_ERR_PROJINACTIVE`.
- Added three new example programs in the folder
"`3do_os:M2_2.7:remote:Examples:Graphics:GraphicsFolio`":
 - `listprojectors`
Lists all the Projectors in the system.

- `listviewtypes`

Lists all the View types in a given Projector.

- `manyview`

Illustrates the creation, display, and manipulation of multiple Views at once.

- To document these changes, we made major changes and additions to graphics folio autodocs. Autodocs were written for the new functions, and the documentation for the Projector Item (listed under Portfolio Item Reference) was significantly amended.

In particular, the Projector autodoc introduces the concept of a “default Projector,” which is the Projector used when you do not explicitly specify one to the folio.

READ THESE DOCUMENTS before using the new Projectors.

6.4.2 Caveats and Known Bugs

- There is a small performance degradation with double buffering. The workaround when performing benchmarking is to use single buffering.

6.5 Graphics State (GState)

6.5.1 Caveats and Known Bugs

- Passing `GS_FreeBitmaps()` an array of uninitialized bitmap Items will cause a “Read from location 0 error”, which will cause M2 to crash. This is likely to happen if the application takes an early exit, calling `GS_FreeBitmaps()`, before `GS_AllocBitmaps()` has been called.

Workarounds:

- Make sure that when your application takes an early exit, your application determines whether `GS_AllocBitmaps()` has been called, and avoids calling `GS_FreeBitmaps()` if `GS_AllocBitmaps()` has not been called.
- Your application can call `GS_FreeBitmaps()` with `numBitmaps = 0`.

6.6 Font Folio

- `SetClipBox()` Fails to Clip

The `SetClipBox()` function, with the enable clipping flag turned on, fails to clip text.

Workaround:

Anytime after calling `SetClipBox()`, call `MoveText()` with the `dx` and `dy` parameters equal to zero. This will not move the text but will activate the clipping.

- Large Fonts May Display Incorrect Letters

Some large fonts (larger than 24 points) can cause a character to be incorrectly rendered. This happens because the logic that determines how many characters can fit in the Texture RAM is off by one character, which causes the last character to overwrite the first character in TRAM. This problem is more likely to occur with larger fonts for which the size of the entire font character data is more than 16 kilobytes.

Partial Workaround:

- Render the corrupted character in a separate TextState, or
- Use SetClipBox() to clip the target text. This often eliminates the incorrect character overwrite but also causes part of the text not to be displayed.
- When DrawText() or DrawString() are used in conjunction with other functions that use destination blending, such as some of the Frame2d functions, the rendered text may have cosmetic errors.

This happens because the font folio does not initialize all of the destination blending registers.

Workaround: Before calling DrawText() or DrawString(), call the CLT function CLT_SetSrcToCurrentDest() to set the source buffer equal to the current destination buffer.

You must do this before each call or series of calls to these functions. You do not have to do it within a series of contiguous calls to one or both of them.

This problem is currently being fixed for a future release.

- Font Folio problem with z buffering.

Text strings with background color do not render when zbuffering is enabled with the CLT_DBZCNTL register set to (0, 0, 0, 0, 1, 1) because the background rectangle and the foreground text are both rendered with the same W value of 0.999998.

(Note that the register setting "0,0,0,0,1,1" is the setting to update the zbuffer and render pixels with a smaller z value, which is the typical way to enable zbuffering.)

There are 2 workarounds for this problem:

- The preferred workaround:
Set DBZ_DBZCNTL to (0, 0, 1, 1, 1, 1); i.e., use DBZCNTL (GS_Ptr(g), 0, 0, 1, 1, 1, 1). This setting updates the zbuffer and renders pixels with a smaller or equal z value.
- The less desirable workaround:
Render the text string without background color.

In a future release, the PenInfo structure will be extended to allow the application to specify different W value for both the background and the foreground of text strings.

6.7 3D Graphics Libraries

6.7.1 Framework

6.7.1.1 Caveats and Known Bugs

- If you are using the built in shapes: cube, cylinder, torus, or sphere in an SDF file, the primitive will receive the wrong material index. This does not affect the SDF files created by our conversion tools, since they never use these primitives.
- If you cannot close a binary SDF file, you should make sure that you have recompiled your code with this 2.7 release.
- Scene_SetTransparent is not working correctly. Currently, setting scene transparency to FALSE turns it on, while setting it to TRUE turns it off. The default is still to have scene transparency off.

6.7.2 Pipeline

6.7.2.1 READ THIS NOW!

- Only one GP is allowed per application.

6.7.2.2 Caveats and Known Bugs

- You cannot use the scaling transforms with lighting, this will lead to an incorrectly illuminated object.
- There is a bug in the 3d pipeline which causes the hardware to hang when rendering faceted pre lit triangle surfaces.

A temporary buffer is allocated for holding the transformed and clipped vertices, initially this is 32k (each vertex can occupy up to 36 bytes). At the start of each primitive the space needed to hold the vertices is compared against the current size and if necessary the buffer is reallocated, however no check is done in the 2.0 or 2.7 version of the software to ensure that this allocation was successful. If the system is running low on memory and the primitive is too large to fit in the current buffer then the machine will hang.

- The only time when a possibly valid return code would be interpreted as TXB_None (-1) is in the case where the user gets the DblDitherMatrixA and DblDitherMatrixB. Values of DitherMatrix are only valid if you actually set them. The user should ignore the non-error code.
- If you are using a macro in the Graphics Lib, do not treat it as a function; a macro to a pointer with ++ will lead to unforeseen results.

6.8 Command List Toolkit (CLT)

6.8.1 Caveats and Known Bugs

- There is a bug when using the Triangle Engine under 2.0 and 2.7. The TE is only reset the first time the triangle engine is used. This means that the first program you run that uses the TE will be running with the TE in a clean state. Subsequent programs will start with the TE left in the state of the last program executed. This means that you must reset all the TE registers to known values or risk unpredictable results.

6.9 Geometry Compiler

6.9.1 Caveats and Known Bugs

- The Geometry Compiler has problems dealing with extremely large files.
- The Macintosh version of the Geometry Compiler behaves unreliably in some environments and on some data files.

6.10 Modeling Package Converters

6.10.1 Strata (ssptosdf)

6.10.1.1 Caveats and Known Bugs

- There is a bug in Strata's texture mapping dialogue; it shows up when you use default textures. You can set horizontal and vertical field "%s", but the number may or may not be output. The work around is to go back and retype a "%" in the field a second time. Change it and hit ok; otherwise you get a zero value and you will get weird texture coordinate number values or your software will crash.

This bug is being fixed for a future release.

6.11 PostPro

6.11.1 READ THIS NOW!

This version does not allow preview of 3D models on the 3DO M2 hardware.

6.11.2 Caveats and Known Bugs

- PostPro only writes the first level of detail when doing a conversion from texture to RGB.
- Texture files with multiple LODs will be shifted to the right by about a quarter inch when saved to a new name from PostPro.
- If PostPro's memory is low, conversions from a texture may result in garbled images. It doesn't take much to max out the PostPro's memory. Therefore, if you are getting garbled images, try increasing PostPro's partition size.

7.0 Video and Data Streaming

7.1 READ THIS NOW!

- Because the audio clock is generated by the CD-ROM clock, the CD-ROM power must be on (and the red indicator light on) in order for audio output, audio timers, and Data Streaming to work. If a program is stuck, try turning on the CD.

7.2 Data Streaming Run-Time Libraries

7.2.1 Data Stream run-time library

7.2.1.1 READ THIS NOW!

- The client application should NEVER call `DSSetPresentationClock()`. (Applications used to call `DSGetClock()` to work around a DataStreamer bug.)

7.2.1.2 Caveats and Known Bugs

- The trace logging facility (for real-time debugging) is set to print its logs to the debugger's Terminal window.
To write the logs to files instead, set the compile-time switch `USE_RAW_FILE` in `subscribertraceutils.c` to "1".
- `DSGoMarker(..., GOMARKER.BACKWARDS)` will return a range error if the stream is beyond the last marker point. This bug is being fixed for a future release.

7.2.2 Subscriber library

7.2.2.1 SAudio Subscriber

Caveats and Known Bugs

- This subscriber supports multiple channels but, because of the limitation of 23 signals per thread, do not use more than four channels simultaneously.
- The sound spooler allocates a signal for each sound buffer allocated. When multiple channels are active at the same time, multiple sound spoolers are created which may cause the 23 signals per thread limitation to be exceeded.

To avoid this, reduce the number of audio buffers allocated per sound spooler by

- Specifying the number of audio buffers as four (the default is eight) when chunkifying your audio data:

```
AudioChunkifier -cs 4 -i sample_audio.aiff -o sample_audio.saudio
```

or by

- Changing the number of buffers using the SAudioTool on a chunkified audio file:

```
SAudioTool -nb 4 -i sample_audio.saudio
```

7.2.2.2 MPEG Audio Subscriber

READ THIS NOW!

An MPEG audio chunk holds one compressed audio frame, so it typically consists of only 1274 bytes including chunk header -- depending on the compression bit rate chosen. The streamer needs a subscriber message for each chunk that is outstanding at a subscriber, plus a couple more per subscriber for other requests. Otherwise, the streamer will run out of subscriber messages and abort stream playback.

Consequently, to play a stream file containing only MPEG audio chunks, either decrease the total stream buffer space (e.g. 20K bytes/buffer * 4 buffers should be plenty), or increase the number of subscriber messages to a few more than the number of chunks that will fit in the total stream buffer space at the same time.

7.2.2.3 EZFlix Subscriber

READ THIS NOW!

- Programs that use the EZFlixSubscriber must link in the EZFlix Decoder library: `libezflixdecoder.a`.

Caveats and Known Bugs

- This subscriber uses an older method of processing, in which it does not begin decompressing a frame until it is almost time to display it. This can cause a delay before the image actually appears on the screen.

7.3 Data Streaming Examples

7.3.1 PlaySA

7.3.1.1 Caveats and Known Bugs

- For MPEG audio stream files, PlaySA can only play all channels simultaneously.

7.3.2 VideoPlayer

7.3.2.1 Caveats and Known Bugs

- This app does not turn on horizontal or vertical interpolation.
This app ought to use the BMTAG_BUMPDIMS tag and the bumped bitmap dimensions instead of the hard-coded dimensions.
- If the image is smaller than 320x240, it places it in the upper left-hand corner of the display rather than centering it.

7.3.3 EZFlixPlayer

7.3.3.1 Caveats and Known Bugs

- When using MemDebug, you may get the following error message when using EZFlixPlayer:

Data Access Fault

This is caused by a memory access bug in the clean-up code of EZFlixPlayer.c. This bug is being fixed for a future release.

7.4 Data Streaming Tools

7.4.1 AudioChunkifier

7.4.1.1 READ THIS NOW!

- Though the AIFC formats provide less compression than MPEG Audio, their run-time decompression consumes virtually no PowerPC CPU cycles, because decompression takes place in the 3DO M2 DSP (or in other 3DO M2 HW in the case of SQS2). MPEG Audio decoding, however, will consume a substantial portion -- 25% or more -- of the CPU.

7.4.1.2 Caveats and Known Bugs

- Currently supports input formats with the following choices of parameters, although the 3DO M2 Audio Folio does not support all possible parameter combinations (see "format restrictions" below). Also, testing priority has gone to 16-bit formats, which we expect to be more widely used than 8-bit formats for reasons of fidelity. See SquashSnd documentation and its usage message for more information on the AIFC compression options.
 - 44100 or 22050 Hz sample rates;
 - stereo or mono;
 - 8 or 16 bit per sample quantization;
 - uncompressed (AIFF) or one of 3 compressed (AIFC) formats:
 - SQS2: 2:1 compression; decompression with M2 HW;
 - CBD2: 2:1 compression; decompression with M2 DSP SW;
 - ADP4: 4:1 compression; decompression with M2 DSP SW.
- Format restrictions:
 - SQS2 compression applies to mono only. (CBD2 works for both mono and stereo.)
 - ADP4 compression applies to mono only.
- SQS2 vs. CBD2 for mono: since SQS2 decoding is HW-based, it takes almost no DSP resources (ticks or program space), but CBD2 may produce slightly better audio fidelity, and CBD2 can be used for stereo.
- ADP4 currently may not be an optimal ADPCM implementation, due to the level of audio distortion perceptible in some material.

7.4.2 MPEGVideoChunkifier

7.4.2.1 Caveats and Known Bugs

- If you omit the `-fps` (frames per second) command-line option, MPEGVideoChunkifier will assume the output frame rate should be that which is specified in the video sequence header of the input MPEG-1 Video

Elementary Stream. This is an enumerated parameter, for which only the following frame rates are defined by the MPEG standard: 23.976, 24, 25, 29.97, 30, 50, 59.94, or 60 fps. Use the `-fps` option to force MPEGVideoChunkifier to make the output frame rate a different value. It will neither add nor subtract frames from the input file, it will only make their frame rate different. This feature is necessary, for example, if you want to use Sparkle to encode QT movies with non-MPEG-standard frame rates (e.g., 12, 15, 20 fps), and if you want to perform the encoding in such a way as to preserve the non-standard frame rate. In this case, you must know the non-standard frame rate of the MPEG Video stream, and specify it to the MPEGVideoChunkifier explicitly, using the `-fps` option.

- MPEGVideoChunkifier is a fairly easy-to-use tool for weaving simple, linear, stand-alone MPEG movie clips. For such simple streams, MPEGVideoChunkifier provides adequate defaults for most of its input parameters, and you need not learn about the details. But for preparing more complex streams, e.g., preparing separate clips which will later be concatenated, or preparing branch points, it is important to understand the command-line parameters in detail.

7.4.3 MPEGAudioChunkifier

7.4.3.1 Caveats and Known Bugs

- Known bug: Invoking MPEGAudioChunkifier with `"-help"` or incorrect arguments will print usage, which may be missing the command name, or may print garbage where the command name should be printed.

7.4.4 EZFlixChunkifier

7.4.4.1 READ THIS NOW!

- Using the `"-h"` and `"-w"` options to crop a movie can cause a crash. This bug is being fixed for a future release.

Workaround: This bug does not happen if all frame dimensions are multiples of 16.

7.4.4.2 Caveats and Known Bugs

- You must specify the desired frame size using the `"-h"` and `"-w"` options or else frame size 256x192 may be assumed.

Note that all frame dimensions must be multiples of 16 because of the aforementioned bug.

- The `"-k"` option, to flag key frames, has been removed.
- The `"-q"` option allows you to set the quality level. Use 25% for an Opera level of quality. Use 75% if you can accept the higher data rate.
- On PowerPCs, garbage characters may be seen in the verbose output when altering the height and width making the output file unusable.

7.4.5 QTVideoChunkifier

7.4.5.1 Caveats and Known Bugs

- QTVideoChunkifier only accepts an EZFlix-compressed QuickTime movie file as input. Since EZFlix is the only SW codec currently supported by 3DO M2 Data Streaming, it's the only type of QT movie file you should be using as input.

-
- Known bug: The QTVideoChunkifier experiences problems on the PowerMac due to an incompatibility between the EZFlix QT Component 1.0b1 and the QuickTime™ 2.0 component. To resolve this problem, either use a utility like "Thing Motel" to install the EZFlix Component, or use a more recent version of QuickTime™ (2.1 or later).

7.4.6 DATAChunkify -- the DATASubscriber's Data Prep Tool

Caveats and Known Bugs

- This tool DOES NOT do data compression. If the `-comp` option is used, the data is assumed to have been compressed with the comp3DO MPW tool, and the chunk header is set so that the DATASubscriber will decompress the data when it is read from the stream.

7.4.7 Weaver

7.4.7.1 Caveats and Known Bugs

- Weaving a stream containing lots of small chunks (e.g., 1 KByte or smaller) with a stream containing fewer large chunks (e.g., half the streamblocksize or larger) can result in a stream where the large chunks fall increasingly behind the small chunks, according to their timestamp order.

This bug does not occur when the larger chunk type can be split across streamblock boundaries.

However, when large chunks cannot be split (e.g. EZFlix video), smaller chunks of other streams (e.g. audio) can still move ahead of them.

If enough small chunks move far enough ahead to make playback unsmooth or blocked up (you can verify this with DumpStream), increase the audio chunk size (e.g. to 4 KBytes or more).

This problem might be severe if EZFlix video were used with MPEG audio, but that combination is not recommended because both those decoders require big percentages of the CPU cycles.

- To set the starting presentation time of an MPEG video elementary stream, use the MPEGVideoChunkifier's `-s` option, NOT the relative starting time in the Weaver-script `"file"` command.

Always set the relative starting time in the Weaver-script `"file"` command to `"0"` for an MPEG video chunk file. The `"file"` command's relative starting time argument can only adjust the delivery timestamps (DTs), not the MPEG video presentation timestamps (PTs). Attempting to use it to adjust MPEG PTs would make playback unsmooth and unsynchronized.

- Weaver scripts should ALWAYS include the `"writestreamheader"` command. Many of the parameters specified by the Weaver script cannot be rendered effective unless the Weaver creates a stream header containing them. It will only do so if this command is included in the script.
- The Weaver's `"writemarkertable"` command adds a spurious first marker, which points to a chunk in the first stream block, e.g. the third header chunk. (Unlike all other markers, this one is not followed by a FILL chunk out to the end of the stream block.) Having an extra marker affects some of the go-marker run-time operations.
- The `"writegotochunk"` command only works when its options argument is `"1"`. It won't accept additional option flags or alternative branch types.

7.4.8 Worksheet and Exercises

On the 3DO M2 2.7 CD, you will find source code for the four Data Streaming example applications, as well as example scripts and data for chunkifying and weaving playable streams. See the folder with pathname:

Examples:M2_2.7:Streaming:

Within this folder, in addition to the four source-code folders, you will find a text file named "ds_examples_readme" and a folder named "Tools_and_Data".

The readme file explains the example apps and streams contained in the overall folder.

The Tools_and_Data folder contains an MPW worksheet named "Video.worksheet", which shows how to use the various chunkifying and weaving tools to create playable streams. Tools_and_Data also contains various elementary streams (MPEG-Video, MPEG-Audio, AIFF audio, and EZFlix video) for use with the worksheet examples.

7.4.8.1 READ THIS NOW!

- The installer will NOT install this folder on your Mac, because of the large size of some of the files.

7.5 Video Tools

7.5.1 3-2 Pulldown, MovieEdit, MovieCompress

7.5.1.1 Caveats and Known Bugs

- MovieEdit: If you attempt to save a movie from the same directory that has the same name as a movie that is also open, both movies will end up invalid. The work around is to close the original movie before replacing it.

7.6 MPEGVideo

7.6.1 Features Added or Changed

- The M2 MPEG Video device no longer clears the sequence header information and state after receiving a flush or detecting a bitstream error. This allows an application to continue decoding from a stream which does not contain frequent sequence header information.
- After detecting a real bitstream error (most likely caused by a media read problem), the M2 MPEG Video device seeks forward to the next picture in the input stream before attempting further decode. This helps to speed recovery.

7.6.2 Bugs Fixed

- In MPEG video driver, fixed a bug in NextStartCode() search that was affecting the Video CD player. The start code search algorithm has been changed to handle cases where start codes cross certain buffer boundaries.
- Spurious MPEG hardware interrupts are generated under certain conditions (e.g., after video bitstream DMA and "picture done" interrupts occur at close to the same time). The device's interrupt handler now handles these interrupts correctly.

This bug and the previously described search bug had been creating symptoms that appeared to be caused by errors in the video bitstream, including "bitstream error" messages.

7.7 MPEG Audio

7.7.1 Bugs Fixed

- The following bug was fixed in the MPEG Audio Decoder folio: If the input packet (or chunk) contained multiple “frames” (MPEG audio access units), the decoder used to emit the same Presentation Time Stamp (PTS) multiple times, which would cause jerky video when the video subscriber tried to sync to the audio. This fix is needed for the VideoCD application, and it will also help if the MPEG AudioChunkifier is ever modified to emit more than one frame per chunk.

8.0 Audio

8.1 Caveats and Known Bugs

- Boards below the Rev G level will not work with M2 Release 2.0 or 2.7.
- Envelope time scaling does not affect envelope loop times set by `AF_TAG_SUSTAINTIME_FP` and `AF_TAG_RELEASETIME_FP`.
- `SampleInfoToTags()` fails to transfer `smpi_Detune` to an `AF_TAG_DETUNE` tag. This also affects `LoadSample()`. The result is that any detune information stored in an AIFF is not copied to the resulting Sample Item.

8.2 Beep Folio

8.2.1 Caveats and Known Bugs

- When using the Beep Folio, you should turn down the amplitude on channels that are stopped. This will prevent DC offsets from those channels from being added to your output which could cause clipping.
- In M2 Release 2.0 or 2.7, if you use the Beep Folio and then want to use the Audio Folio, you must reboot the 3DO Dev Card before switching to the Audio Folio. If you use the Audio Folio and then want to use the Beep Folio, you must reboot the 3DO Dev Card before switching to the Beep Folio.
This will be fixed in a later release, which will support demand loading and unloading of folios.

8.3 Audio folio

8.3.1 Caveats and Known Bugs

- There is a known bug with velocity zones. In order for velocity zones to work properly, a pitch must be specified along with the velocity when calling `StartInstrument()`. Specify the pitch using `AF_TAG_PITCH`.

8.4 Music Library

8.4.1 Sound spooler

8.4.1.1 Caveats and Known Bugs

- Worked around the sample length alignment checks in the audio folio, which could cause perfectly good buffers to be rejected. However, this workaround can permit illegal buffer lengths to be played by the audio folio. A more

reliable system to prevent buffer length alignment problems is being investigated.

Appendix A Examples in Tools Folder

Below is a list of the examples that are shipped with individual tools.

A.1 Audio Examples

A.1.1 test3sf

Source code showing how to call the 3SF sfPlayScore API from a 3DO program.
Located on the CD, inside `":Tools:M2_2.7:Audio:3SF:Examples:"`.

A.2 Development Code Examples

A.2.1 Comm3DOExampleClient 68K

Source code showing how to call the Comm3DO API from a Macintosh program.

Located on the CD, inside

`":Tools:M2_2.7:LowLevel:Comm3DO:Example Client App:"`

A.2.2 Comm3DOExampleClient PPC

Source code showing how to call the Comm3DO API from a Macintosh program.

Located on the CD, inside

`":Tools:M2_2.7:LowLevel:Comm3DO:Example Client App:"`.

A.2.3 C3 Task

Source code showing how to call the Comm3DO API from an M2 program.

Located on the CD, inside

`":Tools:M2_2.7:LowLevel:Comm3DO:Example Client Task:"`.

A.3 Graphics Examples

This is a listing of example and sample code that is to be found in individual tools folders. The path name for these examples is

`":Tools:M2_2.7:Graphics:"`.

A.3.1 M2 Kanji FontViewer

- KFontViewer <fontName>

Demonstrates how to use the Kanji font library to load and display S-JIS encoded text on the 3DO system.

Appendix B Examples in Examples Folder

The 3DO M2 CD contains a central “Examples” folder that contains examples for various M2 components. This folder is in the following location:

```
3do_os:M2_2.7:remote:Examples
```

A list follows of all the example programs in the “Examples” folder.

All the example programs are documented by man pages contained in the *3DO M2 Supplemental Portfolio Reference* unless otherwise noted.

Note that some of the locations given in the man pages are incorrect. The locations given in the following list are correct.

B.1 READ THIS NOW!

- Makefiles for examples have debugging turned on.
- Makefiles should include self as a dependency.

B.2 Audio Examples

The audio examples are contained in

```
3do_os:M2_2.7:remote:Examples:Audio
```

B.2.1 Beep (in Audio:Beep)

- **tb_envelope**
Trigger envelopes using the Beep folio.
- **tb_playsamp**
Play a sample using the Beep Folio.
- **tb_spool**
Spool audio from memory using the Beep folio.

B.2.2 Juggler (in Audio:Juggler)

- **tj_canon**
Uses the juggler to create and play a semi-random canon.
- **tj_multi**
Uses the juggler to play a collection.
- **tj_simple**
Uses the juggler to play two sequences.

B.2.3 MarkovMusic (in Audio:MarkovMusic)

- **MarkovMusic**
Constantly-varying soundtrack. that responds to an environmental influence defined by your application -- in this case the control pad.

B.2.4 Misc (in Audio:Misc)

- **capture_audio**
Record the output from the DSP to a host file.
- **minmax_audio**

Measures the maximum and minimum output from the DSP.

- **playsample**
Plays an AIFF sample in memory using the control pad.
- **simple_envelope**
Simple audio envelope example.
- **ta_attach**
Experiments with sample attachments.
- **ta_customdelay**
Demonstrates a delay line attachment.
- **ta_envelope**
Tests various envelope options by passing test index.
- **ta_pitchnotes**
Plays a sample at different MIDI pitches.
- **ta_sweeps**
Demonstrates adjusting knobs.
- **ta_timer**
Demonstrates use of the audio timer.
- **ta_tuning**
Demonstrates custom tuning a DSP instrument.
- **tone**
Simple audio demonstration.

B.2.4 Score (in Audio:Score)

- **auto_beat**
Automatic rhythm demo that uses many AIFF library samples.
- **playmf**
Plays a standard MIDI file.

Note that playmf is a shell command and is documented in the chapter of the *3DO Supplemental Portfolio Reference* that describes shell commands.
- **playpimap**
Automatic rhythm demo that uses lots of AIFF library samples.

B.2.5 Sound3D (in Audio:Sound3D)

- **ta_bee3D**
Three sounds in a navigable space using 3DSound API.
- **ta_leslie**
Demonstrates directional sound with the 3DSound API.
- **ta_sound3d**
Simple directional sound using 3DSound API.
- **ta_steer3d**
Control a walking man in three-space using the 3DSound API.

- **SeeSound**

Sounds in an audio-visual space, using 3DSound and Framework/Pipeline.

B.2.6 SoundEffects (in Audio:SoundEffects)

- **sfx_score**

Uses libmusic.a score player as a sound effects manager.

- **windpatch**

Creates a "wind" sound effect using the audio folio's patch compiler.

B.2.7 Spooling (in Audio:Spooling)

- **playsf**

Demonstrates how to loop a sound file off disc.

- **ta_spool**

Demonstrates the libmusic.a sound spooler.

- **tsp_algorithmic**

Advanced sound player example showing algorithmic sequencing of sound playback.

Caveats and Known Bugs

When running the audio examples `tsp_algorithmic`, `tsp_rooms`, `tsp_spoolsoundfile`, and `tsp_switcher`, you may notice some stuttering and popping. If you see this behavior, try putting the debugger in Special Mode (Command - =). This bug appears to be a result of the hard drive not delivering data fast enough. Special Mode improves I/O speed and is a more accurate representation of actual CD data access.

- **tsp_rooms**

Room-sensitive soundtrack example using advanced sound player.

- **tsp_spoolsoundfile**

Plays an AIFF sound file from a thread using the advanced sound player.

- **tsp_switcher**

Advanced sound player example that switches between sound based on control pad input.

B.3 Event Broker Examples

The Event Broker examples are contained in

```
3do_os:M2_2.7:remote:Examples:EventBroker
```

- **cpdump**

Queries the event broker and prints out a summary of what's connected to the control port.

- **focus**

Talks to the event broker and switches the focus to a different listener.

- **lookie**

Connects to the event broker and reports any events that occur.

- **luckie**

Uses the event broker to read events from the first control pad.

- **maus**

Uses the event broker to read events from the first mouse.

B.4 FileSystem Examples

The FileSystem examples are contained in

```
3do_os:M2_2.7:remote:Examples:FileSystem
```

- **ls**

Displays the contents of a directory.

- **type**

Type a file's content to the output terminal.

- **walker**

Recursively displays the contents of a directory, and all nested directories.

B.5 Graphics Examples

The Graphics examples are contained in

```
3do_os:M2_2.7:remote:Examples:Graphics
```

B.5.1 CLT (in Graphics:CLT)

- **cltspinclip**

Demonstrates various basic CLT features.

Notes

- Upon exiting the CLTspinclip program a spurious error message is displayed because the program is trying to free a signal that was never allocated. This error does not affect the running of the program and can be ignored. This bug is being fixed for a future release.

- **cltsurface**

Uses the CLT to create and display a 3D surface.

- **gsquare**

This example covers: initialization of the graphics folio and gstate, creating and setting bitmaps, double buffering, loading and display of textures, and rendering geometry, including rectangles, strips and fans.

- **height_field**

Displays a 3-D terrain section whose texture is determined by its height.

This example is self-documenting. Execute it with no parameters to see usage message.

- **scrolling**

Demonstrates multi-level parallax scrolling with the CLT.

B.5.2 Fonts

B.5.2.1 Fixed Bugs

- The following bug was fixed:

The font example makefiles ("fonts.make" and "showmessage.make") located in the directory:

"M2_2.7:3DO:Examples:M2_2.7:Graphics:Fonts:" do not work.
They are backlevel versions and will terminate during the build.

8.4.2.2 Fonts Examples (in Graphics:Fonts)

These examples are self-documenting. Execute them with no parameters to see usage message.

- **bounce**
Shows how to set up a TextState, move, scale, and read a TextState's current position by bouncing the text around the screen.
- **colors**
Shows how to change text colors.
- **newfont**
Runs through a series of font folio tests, and times all the tests.
- **rotate**
Shows how to rotate text on screen.
- **showfont**
Display characters of a font.
- **showmessage**
Outputs any words on the command line of the debugger to the display monitor.
- **testnl**
Tests whether the font folio correctly handles \n characters.

B.5.3 Frame (in Graphics:Frame)

- **anim**
Displays object with keyframe animation using Framework extensions.
- **freespin**
Freely rotate specified objects within a binary SDF.
- **m2perf**
Program to measure the performance of Framework, Pipeline, and the M2
- **newview**
Rotate specified objects within a binary SDF using control pad.
- **sceneperf**
Indicates performance data for rendering a scene.
- **testspin**
Rotate objects within a binary SDF and record performance statistics.
This example is self-documenting. Execute it with no parameters to see usage message.

B.5.4 Frame2d (in Graphics:Frame2d)

- **automapper**
Demonstrates how to map the corners of a sprite.
- **drawlines**

-
- **movesprite**
Demonstrates three methods of drawing lines.
Shows how to move, scale, and rotate a sprite
 - **points**
Shows how to draw points.
 - **rectangles**
Demonstrates two methods of drawing rectangles.
 - **renderorder**
Shows how to alter the render order of sprites using a list-based approach.
 - **simplesprite**
Very simple example of how to draw a single sprite.
 - **spin3d2d**
Shows how to display 3D and 2D graphics simultaneously.

B.5.5 GraphicsFolio (in Graphics:GraphicsFolio)

- **autosizeview**
Illustrates how to create a Bitmap and View automatically sized for the prevailing video mode.
- **basicview**
Illustrates how to create a basic View.
- **listprojectors**
Lists all the Projectors in the system.
- **listviewtypes**
Lists all the View types in a given Projector.
- **manyview**
Illustrates the creation, display, and manipulation of multiple Views at once.

B.6 Kernel Examples

The Kernel examples are contained in

`3do_os:M2_2.7:remote:Examples:Kernel`

- **defaultport**
Demonstrates using a task's default message port to communicate with a child thread.
- **fasttiming**
Demonstrates how to use the high accuracy kernel timing services.
- **memdebug**
Demonstrates the memory debugging subsystem.
- **metronome**
Demonstrates how to use the metronome timer feature.
- **msgpassing**
Demonstrates sending and receiving messages between two threads.

- **signals**
Demonstrates how to use signals.
- **timerread**
Demonstrates how to use the timer device to read the current system time.
- **timersleep**
Demonstrates how to use the timer device to wait for an amount of time specified on the command-line.
- **UserExceptions**
Demonstrates how to trap program exceptions.

B.7 Miscellaneous Examples

The Miscellaneous examples are contained in

`3do_os:M2_2.7:remote:Examples:Miscellaneous`

- **compression** (in `Miscellaneous:Compression`)
Demonstrates use of the compression folio.
- **dbgconsole** (in `Miscellaneous:DbgConsole`)
Example program used to demonstrate 3DODebug functionality.
- **numinfo** (in `Miscellaneous:DebugExamples`)
Example program used to demonstrate 3DODebug functionality.

B.8 MPEG Examples

The MPEG examples are contained in

`3do_os:M2_2.7:remote:Examples:MPEG`

- **photo** (in `MPEG:photo`)
This example application shows how to use the MPEG-Video decoder interface for still-image decompression.
The example loads an MPEG-Video still-image (I-frame) from a file into M2 memory. The file format is simply an MPEG-1 Video elementary stream containing only an I-picture. (It can be created, for example, using Sparkle.) The Photo app then opens the MPEG-Video device for I-picture-only decoding, so that the device allocates no reference-frame buffers in main memory. The device returns the decompressed I-frame to the Photo app as soon as it is finished decoding, without the pipeline delay incurred with full I/P/B movie decoding.
- **playfast** (in `MPEG:playfast`)
Demonstrates the power of the 3DO M2 MPEG-1 Video decoder, and also serves as example code for the MPEG-1 Video device driver.
Works by loading into 3DO M2 RAM as much of an input MPEG-1 Video elementary stream as it has space for, and then playing this stream repeatedly in a loop.
This example does not use the Data Streaming interface to M2's MPEG-1 Video decoder.

B.9 Data Streaming Examples

This section summarizes and gives pointers to the Data Streaming example applications. For more detail and notes on most of these applications, see “Data Streaming Examples” on page 18.

The Data Streaming examples are contained in

`3do_os:M2_2.7:remote:Examples:Streaming`

- **DataPlayer** (in `Streaming:DataPlayer`)

Uses the `DATASubscriber` to “play” a stream containing DATA chunks. The application simply prints statistics about data blocks as they are delivered, but it exemplifies how to set up, interact with, and destroy the `DATASubscriber`.

- **EZFlixPlayer** (in `Streaming:EZFlixPlayer`)

Plays an EZFlix (3DO M2 SW-decodable video) stream, with or without a synchronized native 3DO M2 audio stream.

- **PlaySA** (in `Streaming:PlaySA`)

Plays one or more native 3DO M2 audio streams woven into a single multiplexed stream.

- **VideoPlayer** (in `Streaming:VideoPlayer`)

Plays an MPEG-1 Video stream, with or without a synchronized native 3DO M2 audio stream.



3DO M2 Mercury Programmer's Guide

Version 2.7 – June1996

Copyright © 1996 The 3DO Company and its licensors.

3DO and the 3DO logos are trademarks and/or registered trademarks of The 3DO Company. All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

Preface

About This Book	MER-vii
About the Audience	MER-vii
System Requirements	MER-vii
How This Book Is Organized	MER-vii
Typographical Conventions	MER-viii
Related Documentation	MER-viii

1

Introducing Mercury

Understanding Mercury	MER-2
Pipeline Object Descriptors (PODs)	MER-2
How PODs Are Used in the helloworld Program	MER-2
Why PODs Are Important	MER-3
Creating an Application with Mercury	MER-3
The Mercury API	MER-4
Mercury's Four Core Functions	MER-4
Other Mercury Functions	MER-4
Data Structures Used in Mercury	MER-6
Mercury Instruction Caching	MER-6
Mercury's Library Hierarchy	MER-6
How Mercury Instruction Caching Works	MER-7
Example: Building a Mercury Application	MER-7
Inside Mercury	MER-9
The Sort Stage	MER-10
The POD Initialization Stage	MER-10
The Vertex Pipeline Stage	MER-11

The Triangle Assembly Stage	MER-11
The CloseData Structure	MER-12
Mercury Geometry	MER-13
Geometry Structures Used in Mercury	MER-13
The PodGeometry Structure	MER-14
Example: Using Mercury Geometry to Construct an Object	MER-14
Drawing Clockwise and Counter-Clockwise Triangles	MER-17
Flags Used in the PodGeometry Structure	MER-20
The Pod Structure	MER-21
Mercury Lighting	MER-24
Mercury's Light Functions	MER-24
Light Lists	MER-25
Lighting Structures and Material Structures	MER-25
Using Lighting Structures and Material Structures	MER-27
The Calling Sequence of Lighting Operations	MER-27
Varieties of Lighting Used in Mercury	MER-28
The Two Stages of a Mercury Lighting Routine	MER-28
Example: Using Lighting Data	MER-29
Camera Operations in Mercury	MER-30
The M_SetCamera Function	MER-30
The Matrix_Perspective Function	MER-30
How a Skew Matrix Works	MER-31
A Camera Matrix in World Space	MER-31
Constructing the Skew Matrix	MER-32
Mercury Texture-Mapping	MER-33
Summary	MER-34

2

Using Mercury

Anatomy of a Mercury Application	MER-36
About the helloworld Program	MER-36
Preparing to Write a Mercury Game	MER-37
Creating and Initializing a GState Object	MER-38
Initializing Mercury in the helloworld Program	MER-40
Creating and Rendering Frames	MER-42
Writing Your Game Code	MER-43
Performing Bounds-Testing	MER-43
Displaying Your Game's PODs	MER-44
The Game-Play Section of the helloworld Program	MER-44
Cleaning Up	MER-45
Summary	MER-46

A

Parts of the Mercury Engine

The Core Functions	MER-47
M_Init	MER-47
M_End	MER-48

M_Sort	MER-48
M_Draw	MER-48
Utility Functions.....	MER-49
M_Prelight	MER-49
M_BoundsTest	MER-49
M_LoadPodTexture	MER-49
M_SetCamera	MER-50
Matrix_Perspective	MER-50
M_Draw. . . Functions	MER-50
M_DrawDynLit	MER-51
M_DrawDynLitTex	MER-51
M_DrawDynLitTrans	MER-51
M_DrawPreLit	MER-51
M_DrawPreLitTex	MER-51
M_DrawPreLitTrans	MER-51
M_Light. . . Functions.....	MER-51
Initializing Mercury's Light Functions	MER-52
Mercury's Light Functions Listed and Described	MER-52
Texture-Blender and Destination-Blender Setup Macros	MER-57
M_TBNoTex	MER-57
M_TBLitTex	MER-57
M_TBTex	MER-57
M_TBFog	MER-57
M_DBNoBlend	MER-57
M_DBFog	MER-57
M_DBSpec	MER-58
M_DBTrans	MER-58
M_DBInit	MER-58
Case Setup Functions	MER-58

B**Matrix Calls****C****Vector Calls**

Preface

About This Book

This manual describes the 3DO M2 Mercury rendering engine and shows how you can use it to create M2 applications.

About the Audience

This manual is written for title developers. The information presented and the level of description is based on the assumption that you are familiar with both the C programming language, and with three-dimensional graphics.

System Requirements

To use Mercury, you should have access to a working 3DO M2 development system and to an Apple Power Macintosh or Macintosh Quadra running Operating System 7.5 or higher. You must also have access to a 3DO M2 system installed in accordance with the installation procedures described in the *3DO Development Environment Installation Guide*.

How This Book Is Organized

Chapter 1, "Introducing Mercury " provides a brief overview of Mercury, describes its components, and lists and describes the steps that are needed to create an M2 application using Mercury.

Chapter 2, "Using Mercury " presents and describes an example program that shows how to develop a simple Mercury application.

Appendix A, "Parts of the Mercury Engine" lists and describes the C-language function calls provided in the Mercury API.

Appendix B, "Matrix Calls" lists and describes Mercury matrix functions.

Appendix C, "Vector Calls" lists and describes Mercury vector functions.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>Scene_GetStatic(scene)</code>
procedure name	<code>Char_TotalTransform()</code>
new term or emphasis	In M2, <i>characters</i> are objects that can be displayed on the screen.
file or folder name	The <i>remote</i> folder, the <i>demo.scr</i> file.

Related Documentation

To understand the material in this manual, and to use Mercury in your application, it is necessary to be familiar with the material in the following 3DO M2 manuals:

- ◆ *Getting Started With 3DO M2 Release 2.0*
- ◆ *The 3DO M2 Debugger Programmer's Guide*
- ◆ *The 3DO M2 Graphics Programmer's Guide*
- ◆ *The 3DO M2 Command List Toolkit manual*

In addition, the following books are recommended as valuable sources for information on computer graphics and 3D graphics:

- ◆ Foley, van Dam, Feiner, and Hughes. *Computer Graphics Principles and Practice*: Addison-Wesley, 1990
- ◆ Neider, Davis, and Woo. *Open GL Programming Guide*: Addison-Wesley, 1993
- ◆ OpenGL Architecture Review Board: *Open GL Reference Manual*: Addison-Wesley, 1993.

Introducing Mercury

Mercury is a 3D graphics pipeline that is optimized to run in the 4K instruction cache of the Power PC (PPC) 602 CPU that drives M2. Although Mercury is designed to be used by programs written in C or C++, it runs at a speed approaching that of assembly language because all its internal procedures are written in assembly language and pass arguments via registers.

Unlike a conventional C-language API that repeatedly moves data back and forth between CPU registers and a stack, Mercury operates as a simple pipeline that maintains data in a compact data area, paying special attention to data cache alignment.

Because of its efficient and optimized bare-bones design, Mercury can be easily extended and modified to meet the individual needs of each title you develop. Once you understand Mercury's functional interface, you can modify or even replace its functions to suit the requirements of your title and your own personal preferences.

This chapter describes the Mercury rendering engine and explains how to get started using it. To demonstrate basic features of the Mercury system, the chapter uses an example program named *helloworld*, which you can find in the Examples folder on your distribution CD-ROM.

This chapter focuses on the following major topics:

Topic	Page
Understanding Mercury	2
Creating an Application with Mercury	3
The Mercury API	4
Mercury Instruction Caching	6
Inside Mercury	9
The CloseData Structure	12
Mercury Geometry	13
Mercury Lighting	24
Mercury Texture-Mapping	33
Camera Operations in Mercury	30
Summary	34

Understanding Mercury

The heart of the Mercury system is a rendering engine that can be accessed from C or C++ applications. To use the Mercury rendering engine, you provide it with a linked list of objects called pipeline object descriptors (PODs). From this linked list, Mercury produces a command list that is then sent to the M2 Triangle Engine for further processing. (For more information about command lists and the Triangle Engine, see the *3DO Command List Toolkit* manual.)

Pipeline Object Descriptors (PODs)

When you start creating titles using Mercury, you will become very familiar with pipeline object descriptors because Mercury is a POD-based system. In Mercury, PODs are objects that contain pointers to textures, geometries, transformation matrices, and a wide variety of objects used in rendering, texturing, lighting, and manipulating properties of materials.

Mercury treats each POD in a 3D scene as a geometry entity that is transformed independently. Every POD used in a scene is associated with a texture, geometry, a transform, and a list of the lights that are used to illuminate the POD. To create and implement an application using Mercury, you define a set of PODs that meet your application's requirements and then manipulate them using C-language calls and macros provided by Mercury.

How PODs Are Used in the helloworld Program

The *helloworld* sample program shows how PODs can be used in a Mercury program. *helloworld* is a simple application that uses just two PODs: one to create and manage a model, and one to create and manage a background. the model POD is named `firstPod`, and the background POD is named `gCornerPod` (because the program displays a fighter aircraft floating in a box that resembles the corner of a room).

The `gCornerPod` object used in the *helloworld* program is defined as follows in a file named *data.c*:

```
Pod gCornerPod = {
    /* uint32 flags */                0,
    /* struct Pod *pNext */          NULL,
    /* void (*pcase)(CloseData*) */  M_SetupPreLit,
    /* struct PodTexture *ptexture */ NULL,
    /* struct PodGeometry *pgeometry */ &gCornerGeometry,
    /* Matrix *pmatrix */             &gCornerMatrix,
    /* uint32 *plights */             gCornerLightList,
    /* uint32 *puserdata */          NULL,
    /* Material *pmaterial */         &gCornerMaterial
};
```

Why PODs Are Important

PODs are vital in a Mercury application because almost all operations performed in Mercury depend upon them. In a Mercury-based game, you use PODs to compute bounding boxes, perform lighting operations, and display all backgrounds and game characters. You'll find POD objects referred to repeatedly throughout this chapter and throughout Chapter 2, "Using Mercury."

For more details about how PODs are constructed and how they are used in Mercury programs, see the section headed "Mercury Geometry" on page 13.

Note: *The rendering performance that you can achieve with Mercury varies with factors such as the size of the POD, the number of different textures used in a scene, and both the number of lights used and the kind of lighting that is created. The amount of shared information among PODs, such as textures, also affects rendering performance.*

Creating an Application with Mercury

Although Mercury can be used to create many different kinds of games, the process of developing an application using Mercury can be broken down into just a few general steps. Those steps can be summarized as follows.

1. Perform the initialization operations your application requires. Initializing a Mercury-based application includes the following steps:
 - ◆ Creating and initializing a `GState` object
 - ◆ Initializing Mercury
 - ◆ Creating and setting up a camera.

All three of these operations are covered in more detail in Chapter 2, "Using Mercury."

2. Perform the work that is needed to create and render each frame in your game. This section of your application contains the code that performs the standard per-frame operations, such as frame-buffering and page-flipping. Code in this section also initializes the M2 destination blender.
3. Write the code that is required to run your game. In this section, you update all POD transforms, and you create and display the specific PODs that are needed at runtime in each stage of your game. Typically, code executed in this section of a game calls various Mercury and `GState` functions to perform such

operations as displaying rendered images (such as 2D images), displaying 3D Mercury PODs, and handling user input.

4. Write code that shuts your game down and performs all needed cleanup functions when the user terminates the program.

The sequence of operations described in this list is examined in Chapter 2, "Using Mercury."

The Mercury API

The Mercury API (application programming interface) is made up of functions that can be called from C-language applications, and data structures that are accessed by those functions. This section briefly describes the functions and data structures provided in the Mercury API. The functions and data structures introduced in this section are covered in more detail under separate headings later in this chapter and in Appendix A, "Parts of the Mercury Engine."

Mercury's Four Core Functions

Four of the most important function calls provided in the Mercury API are `M_Init`, `M_End`, `M_Sort`, and `M_Draw`. These four functions are vital to the operation of the Mercury system. They perform the following operations:

- ◆ `M_Init` initializes the system for rendering. Your application must call `M_Init` before it calls any other Mercury functions.
- ◆ `M_End` frees up all memory allocated in `M_Init`, including the memory allocated to the `CloseData` structure passed into this routine.
- ◆ `M_Sort` sorts the linked list of PODs that your application provides to Mercury. Mercury must sort the list of PODs that you provide before it can process them.
- ◆ `M_Draw` takes your application's list of PODs as input and generates an M2 command list that can then be used for rendering your PODs on the screen.

Mercury's core functions are described in more detail in "The Core Functions" on page 47.

Other Mercury Functions

Along with the four core functions described under the previous heading, Mercury provides a number of other important functions that perform more specific kinds of operations. These additional functions can be broken down into the following categories:

- ◆ *M_Draw . . . functions*, which are called to manage Mercury's drawing operations. The name of each draw function begins with the letters `M_Draw`. For example, the `M_DrawDynLit` function dynamically lights triangles according to the list of lights in each POD. In this book, this group of functions is referred to generically as Mercury's `M_Draw . . . functions`. These `M_Draw . . . functions` should be distinguished from Mercury's main `M_Draw` function (see "Mercury's Four Core Functions," preceding) which is not followed by an ellipsis (. . .) when its name appears in this volume. For more information on Mercury's `Draw . . . functions`, see "M_Draw . . . Functions" on page 50.

- ◆ *M_Light... functions*, which are used to control lighting in Mercury applications. All of Mercury's light functions have names that begin with the letters `M_Light`. For example, the `M_LightFog` function computes an alpha value that can be used for a fogging effect. In this book, this group of functions is referred to collectively as the `M_Light... functions`. For more information on the `M_Light... functions`, see "M_Light... Functions" on page 51.
- ◆ *Matrix functions*, which are used to construct and manipulate matrices. Mercury's matrix functions are listed and described in Appendix B, "Matrix Calls."
- ◆ *Vector functions*, which are used to construct and manipulate vectors. the vertex functions are listed and described in Appendix C, "Vector Calls."
- ◆ *Utility functions*, which perform operations related to lighting, bounds-testing, cameras, and textures. Utility functions include `M_PreLight`, `M_BoundsTest`, `M_LoadPodTexture`, `M_SetCamera`, `M_Matrix_Perspective`, and others. `M_PreLight` computes lighting values, `M_BoundsTest` tests bounding boxes to determine whether they are visible, and `M_LoadPodTexture` loads textures. `M_SetCamera` takes a normal camera matrix in world space and multiplies it with a specially formatted skew matrix to form Mercury's internal camera skew matrix (see "Constructing the Skew Matrix" on page 32). The `Matrix_Perspective` function is used to prepare a specially formatted skew matrix that Mercury uses (see "The Matrix_Perspective Function" on page 30).
- ◆ *Texture-blending and destination-blending setup functions*, which initialize Mercury's texture blender and destination blender. Each texture-blending function and destination-blending function has a name that begins with the letters `M_TB` (for texture-blending functions) or `M_DB` (for destination-blending functions). For example, the `M_DBInit` function initializes the destination blender state for use with other `M_DB` calls.
- ◆ *Case setup functions*: In Mercury, the *case* of a POD is the combination of the state in the Triangle Engine and the draw function used to construct the triangle commands. Each case used in Mercury is equipped with a small *case setup function* that is called when the first POD of the case is encountered—that is, when the `samecaseFLAG` is not set in the POD. When a case setup function is called, it places the draw function used by all PODs in the `CloseData` structure. Then it loads a command list into the Triangle Engine. Mercury provides case functions for 15 predefined cases. These functions are listed in "Case Setup Functions" on page 58, along with a table that shows all legal combinations of case setup functions.

For more detailed descriptions of the functions described in the preceding list, see Appendix A, "Parts of the Mercury Engine."

Note: It is important to distinguish between Mercury's `M_Draw` function and all other Mercury functions that begin with the letter's `M_Draw`. There is only one `M_Draw` function, but there are a number of other functions that begin with the letters `M_Draw`—for example, `M_DrawDynLit`, `M_DrawPreLit`, and `M_DrawDynLitTex`. In this book, Mercury's main `M_Draw` function is referred to simply as `M_Draw`, while the other functions that begin with the letters `M_Draw` are referred to collectively as the `M_Draw... functions`. Also note that Mercury has a set of functions that begin with the letters `M_Light`. This group of functions is referred to collectively as Mercury's `M_Light... functions`.

Data Structures Used in Mercury

The Mercury engine makes extensive use of a small set of data structures. Many of the function calls used in Mercury work by setting attributes that are defined as fields in these data structures, and many other Mercury calls return the values of the fields in these data structures. The data structures used most often in Mercury applications are:

- ◆ The `PodGeometry` structure is a geometry data format used in all of Mercury's draw routines. It is made up of a 44-byte header, a set of values used to define vertices, a set of vertex indices, and a set of texture-map coordinates. For more information on the `PodGeometry` structure, see "The `PodGeometry` Structure" on page 14.
- ◆ The `Pod` structure is the highest-level data structure used in Mercury. The header section of the `PodGeometry` structure, described above, is a `Pod` structure. The job of the `Pod` structure is to associate a POD's geometry with a Matrix, a light list, a surface Material, one or more textures, and the code which Mercury will call to actually render the object. The `Pod` structure is described in more detail in "The `Pod` Structure" on page 21.
- ◆ The `CloseData` structure stores all input to the Mercury pipeline. Mercury also uses the `CloseData` structure as a repository for intermediate calculations and temporary data. For more details about the `CloseData` structure, see "The `CloseData` Structure" on page 12.
- ◆ The `Material` structure contains values that are used to calculate the base, diffuse, shine, and specular colors of each object used in Mercury. The `Material` structure is examined more closely in "The `Material` Structure" on page 26.

Mercury Instruction Caching

Mercury is designed to use the Power PC's 4K instruction cache in the most efficient way possible. For example, to ensure that different kinds of routines do not conflict with each other at runtime, Mercury places them in different libraries that can be linked in a specific order when an application is being built. If an application follows this recommended linking sequence at link time, the result is faster execution at runtime.

This section explains how you can take advantage of the caching optimizations built into Mercury's library structure and use those optimizations to design faster-running games.

Mercury's Library Hierarchy

If you open the Mercury folder provided on your distribution CD-ROM, you'll see that Mercury's code is placed in the folders shown in Table 1-1.

Table 1-1 *Contents of the Mercury libraries*

Library	Contents
<i>libmercury1</i>	Code that implements the <code>M_Draw</code> routine
<i>libmercury2</i>	Mercury's <code>M_Light...</code> routines
<i>libmercury3</i>	Mercury's <code>M_Draw...</code> (rendering) routines

Table 1-1 Contents of the Mercury libraries

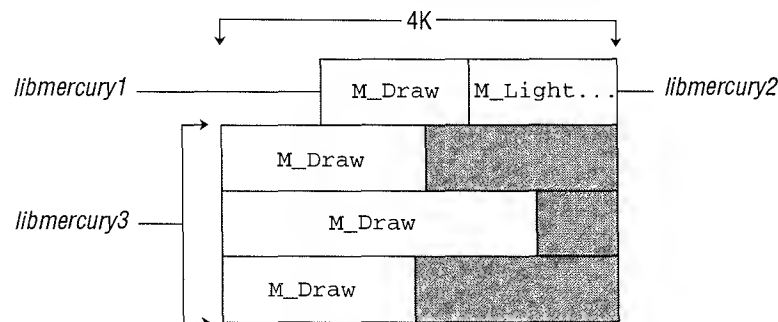
Library	Contents
<i>libmercury1</i>	Code that implements the M_Draw routine
<i>libmercury4</i>	Miscellaneous routines
<i>libmercury_setup</i>	Mercury's setup routines
<i>libmercury_util</i>	Utility routines

To make sure that the routines in your application cache data and retrieve it as efficiently as possible, it is important to link *libmercury1*, *libmercury2*, and *libmercury3* in the same order in which they are numbered—that is, to link *libmercury1* first, followed by *libmercury2* and then by *libmercury3*.

There is no particular significance in the order in which you link your application with *libmercury4* and *libmercury_util*. That's because *libmercury4* and *libmercury_util* contain only utility routines and other miscellaneous functions.

How Mercury Instruction Caching Works

Figure 1-1 shows why linking order is important in Mercury-based games. Notice that instructions in Mercury's *libmercury1*, *libmercury2*, and *libmercury3* libraries are cached into different areas of the 4K instruction cache provided in the Power PC.

**Figure 1-1** Mercury Instruction caching

As you can see, the M_Draw and M_Light... instructions (implemented in *libmercury1* and *libmercury2*) are cached into memory first, followed by the application's M_Draw... functions and their arguments, which are implemented in *libmercury3*.

Because Mercury instructions are cached in this sequence, you can speed up their processing by simply making sure that they are linked in the same order.

Example: Building a Mercury Application

The order in which you link your game's setup routines is also significant. Most games don't use every setup routine that Mercury provides, but the setup routines that you do use should be linked to your application in the same order shown in Example 1-1 (the makefile used by the *bigcircle* example program).

Example 1-1 *Linking Mercury Setup Routines*

```
#    @(#) bigcircle.make 96/03/06 1.2

LIBMERCURY = 0
-1 :::lib:libmercury1:objects:libmercury1 0
-1 :::lib:libmercury2:objects:libmercury2 0
-1 :::lib:libmercury3:objects:libmercury3 0
-1 :::lib:libmercury4:objects:libmercury4 0
-1 :::lib:libmercury_utils:objects:libmercury_utils
LIBSETUP = :::lib:libmercury_setup:objects:

OBJECTS = 0
:objects:game.c.o 0
:objects:gamedata.c.o 0
:objects:gamedata2.c.o 0
:objects:mainloop.c.o 0
:objects:icacheflush.s.o 0
::helloworld:objects:bsdf_read.c.o 0
::helloworld:objects:data.c.o 0
::helloworld:objects:filepod.c.o 0
::helloworld:objects:graphicsenv.c.o 0
::helloworld:objects:scalemodel.c.o 0
::helloworld:objects:tex_read.c.o 0
0
{LIBSETUP}M_SetupDynLit.c.o 0
{LIBSETUP}M_SetupDynLitFog.c.o 0
{LIBSETUP}M_SetupDynLitTex.c.o 0
{LIBSETUP}M_SetupDynLitSpecTex.c.o 0
{LIBSETUP}M_SetupDynLitFogTex.c.o 0
{LIBSETUP}M_SetupPreLit.c.o 0
{LIBSETUP}M_SetupPreLitFog.c.o 0
{LIBSETUP}M_SetupPreLitTex.c.o 0
{LIBSETUP}M_SetupPreLitFogTex.c.o 0
{LIBSETUP}M_SetupDynLitTrans.c.o 0
{LIBSETUP}M_SetupDynLitTransTex.c.o 0
{LIBSETUP}M_SetupDynLitFogTrans.c.o 0
{LIBSETUP}M_SetupPreLitTrans.c.o 0
{LIBSETUP}M_SetupPreLitTransTex.c.o 0
{LIBSETUP}M_SetupPreLitFogTrans.c.o

# Choose one of the following:
DEBUG_OPTIONS = -g # For best source-level debugging
# DEBUG_OPTIONS = -XO -Xunroll=1 -Xtest-at-bottom -Xinline=5

asm = ppcas

# variables for compiler tools
defines= -D__3DO__ -DOS_3DO=2 -DNUPPSIM -DMACINTOSH

dcopts= 0
-c -Xstring-align=1 -Ximport -Xstrict-ansi -Xunsigned-char 0
{DEBUG_OPTIONS} 0
-Xforce-prototypes -Xlint=0x10 -Xtrace-table=0

appname= bigcircle
```

```
LIBS = -lclt -lspmath -leventbroker -lc @
      {LIBMERCURY}

MODULEDIR= "{3doreMOTE}"System.m2:Modules:
BOOTDIR= "{3doreMOTE}"system.m2:Boot:
MODULES= "{MODULEDIR}"graphics "{MODULEDIR}"gstate "{MODULEDIR}"fsutils
"{MODULEDIR}"iff "{BOOTDIR}"kernel "{BOOTDIR}"filesystem
INCLUDEDIR = :::include:

{appname} ff {appname}.make {OBJECTS}
    link3do -r -D -Htime=now -Hsubsyz=1 -Hname={appname} -Htype=5 -Hstack=32768
@
    -o {appname} @
    {OBJECTS} @
    -L"{3dolibs}{m2librelease}" @
    {MODULES}      # @
    {LIBS}          # @
    > {appname}.map# generate extra map info and redirect from dev:stdout
to a map file
    setfile {appname} -c '3DOD' -t 'PROJ' # Set creator and type so
                                         # icon appears
    duplicate {appname} -y {3doreMOTE}# copy the goodies to /remote

# files in the object's sub-dir are dependent upon sources in the current dir
:objects: f :

# .c.o files are dependent only upon the .c source
.c.o f .c
    dcc -I{INCLUDEDIR} -I::helloworld: {defines} {dccopts} {depDir}{default}.c
-o {targDir}{default}.c.o

.s.of .s
    {asm} -I {INCLUDEDIR} {depDir}{default}.s -o {targDir}{default}.s.o
```

Inside Mercury

Because Mercury is designed to function as a pipeline, it performs its internal processing in an order that is predetermined and quite rigid. The sequence in which Mercury performs its processing operations during the execution of an application can be broken down into four stages.

The first stage of processing, called the *sort stage*, is a processing step in which PODs are sorted to optimize the use of TRAM and data caching. To begin the sort stage of processing, you call the `M_Sort` function.

The second, third, and fourth stages of internal processing are called the *POD initialization stage*, the *vertex pipeline stage*, and the *triangle assembly stage*. Mercury carries out all three of these processing stages automatically when you call the `M_Draw` function, which is defined in the `M_Draw.c` file. These three processing stages can be summarized as follows:

- ◆ In the *POD initialization stage*, Mercury performs initialization chores required by each POD. In this stage, each of the following operations takes place:
 - ◆ Concatenation between the camera matrix (see ***) and the objects matrix
 - ◆ Checking the object for screen visibility
 - ◆ Initialization of lighting.
- ◆ In the *vertex pipeline stage*, the following operations take place:
 - ◆ Each vertex in each POD is transformed into screen coordinates
 - ◆ If clipping is applied, each vertex is tested for screen visibility
 - ◆ Each vertex is lighted (when POD lighting is used).
- ◆ The starting point of the triangle assembly stage varies from application to application and from case to case. In the *triangle assembly stage*, the following operations are performed:
 - ◆ Vertices are cull-tested
 - ◆ Texture coordinates are scaled
 - ◆ M2 vertex commands are created and written.

Each of the four stages of Mercury processing is described in more detail under the headings that follow.

The Sort Stage

When Mercury receives a linked list of PODs from an application, the first thing that happens is that the list is sorted. This stage of processing is called the *sort stage*. To begin this stage of processing, you call the `M_Sort` function, which is implemented in the file `sort.s`.

To sort a POD list, Mercury uses six POD fields: `pcase`, `ptexture`, `pgeometry`, `pmatrix`, `plights`, and `puserdata`.

Using these six fields, Mercury sorts each POD list in ascending order. First, all PODs with the same value in the `pcase` field are sorted according to the address `pcase`. Then all PODs with the same `pcase` values are sorted according to the value in the `ptexture` field, and so on.

To perform its sorting operations, Mercury uses a merge-sort technique that strives to keep all PODs in the data cache.

During this sort stage, Mercury sets the `samecaseFLAG` and `sametextureFLAG` bits in the `flag` field of each POD. Consequently, your application should call Mercury's sort code at least once to make sure that these flags are set.

The POD Initialization Stage

When you pass a list of PODs to Mercury, a set of initialization operations must be performed on each POD before further processing can begin. The stage of operation in which these initialization tasks are performed is called the *POD initialization stage*.

To begin this stage of processing, an application calls the `M_Draw` function, which is defined in the `M_Draw.s` file. Mercury then starts performing a set of initialization operations that can be broken down into the following substeps:

1. If the *case* of a POD is different from the case of the previous POD—that is, if the POD's `samecaseFLAG` is false—Mercury executes a block of code called *case code* to set up the POD's `CloseData` structure. (The `CloseData` structure is introduced in “The `CloseData` Structure” on page 12. The *case* of a Mercury POD is the combination of the state in the Triangle Engine and the draw function used to construct the triangle commands. Cases are described in more detail in “Other Mercury Functions” on page 4.)
2. If the texture is different from the texture in the previous POD—that is, if the POD's `sametextureFLAG` is false—Mercury's texture load routine is called to load the PIP and the TRAM with the texture. (For more about PIP tables and texture mapping, see the *3DO Command List Toolkit* manual.)
3. Compute the POD's total transform by concatenating the POD's *pmatrix* transform with the *fcamskewmatrix* transform in `CloseData`.
4. Test the POD's bounding box for visibility on the screen.
5. Invert the POD's total transform.
6. Call the initialization routine for each light source used by a POD.

The Vertex Pipeline Stage

The *vertex pipeline stage* is the stage of operations in which your application calls Mercury's `M_Draw...` and `M_Light...` utility functions. The `M_Draw...` functions and the `M_Light...` functions are implemented in *.s* files with names that begin with the letters *M_Draw* and *M_Light*, which are in the *libmercury2* and *libmercury3* folders.

Code that is executed during the vertex pipeline stage of processing varies from application to application and from POD case to POD case, but the steps that are used to process the code are always the same during this stage of processing. Specifically each vertex in the POD is:

1. Transformed into screen coordinates
2. Lighted (for lit cases only).

To optimize processing during the vertex pipeline stage, Mercury processes vertices in pairs, two vertices at a time. That means, of course, that an even number of vertices must always exist. If an odd number of vertices is present, your application must create a dummy vertex by simply making a duplicate of the last one.

When vertices are lighted, Mercury's lighting code exits early if both normal vectors in the pair point away from the light. So vertices that have similar normals should be grouped together in pairs.

The Triangle Assembly Stage

The last stage of Mercury's pipeline processing is the stage in which M2 vertex commands are created and written. This stage of processing is called the *triangle assembly stage*. The code that is executed in this stage varies from case to case. The names of the files that implemented the code also vary from case to case. The entry point associated with this stage is case-specific as well.

Again, however, the steps that are used to process code during this stage do not vary. During the triangle assembly stage, the following operations occur:

1. The texture coordinates are multiplied by $1/w$ (for textured cases only).
2. The texture coordinates are scaled in accordance with the size of the texture.
3. Each triangle is tested for back face rejection.
4. The vertices of each triangle are clipped to the screen boundary and clipped in hither.
5. The indices that form strips and fans are converted to M2 strip and fan commands.

The CloseData Structure

In Mercury, a structure called the `CloseData` structure contains all input to the graphics pipeline. The `CloseData` structure is also used as a repository for intermediate calculations and temporary data.

Many of the function calls used in Mercury access fields that are defined in the `CloseData` structure. Mercury uses several `CloseData` fields in its rendering code, and those fields must be set up by the calling program before the program calls the `M_Draw` function to render its images on the screen.

Table 1-2 lists the fields of the `CloseData` structure that must be filled in before your application calls `M_Draw`.

Table 1-2 *CloseData Fields that Your Application Must Supply*

Field	Contents
float fwclose	w value for the hither plane
float fwfar	w value for yon plane
float fscreenwidth	screen width in pixels
float fscreenheight	screen height in pixels
uint32 fogcolor	the packed fog color (output by <code>M_PackColor</code>) only needed if using fog
uint32 srcaddr	address of the pixels in the current frame buffer only needed if using transparency
uint32 depth	depth, in bits, of the current frame buffer, only needed if using transparency
float fcamx, fcamy, fcamz	position of the camera in world coordinates
Matrix fcamskewmatrix	a matrix that transforms world coordinates into screen coordinates

Mercury Geometry

Mercury 3D geometry can be defined in terms of independent rigid bodies, each possessing a transformation that orients and places it in world coordinates. The observer is moved around by a mechanism called the *camera transformation*, which maps the world coordinate system into the screen coordinate system.

Even though the transformations of individual pieces used in a game may be hierarchically related, the vertices of the geometry used to render those pieces must ultimately be transformed into the coordinate system used by M2. Consequently, Mercury splits up the CPU cycles of the 602 into the following tasks:

- ◆ Game play
- ◆ Transforming and lighting vertices
- ◆ Assembling the triangles formed between the vertices into M2 commands.

Note: *In the preceding list, the term game play includes such tasks as updating the transformations of individual objects in the scene and updating the camera every frame.*

Geometry Structures Used in Mercury

Mercury uses a geometry model called the `PodGeometry` data structure in all its draw routines. The `PodGeometry` structure provides all of Mercury's components with a consistent structure for representing geometry objects, and the result of this consistency is a combination of high performance and flexibility.

Encapsulating geometry objects into `PodGeometry` structures has numerous advantages. For example, when a model is to be rendered, it does not matter whether the model is textured or untextured, whether it is pre-lit or dynamically lit, or whether it is transparent or opaque. The underlying data structure used to render the model is the same.

Other advantages of using `PodGeometry` structures are:

- ◆ By calling a function named `M_Prelight`, you can convert models converted from dynamically lit models (with normals) to pre-lit models (with colors).
- ◆ You can easily change the texture used to render a model because Mercury uses normalized *u* and *v* coordinates.
- ◆ Vertices and colors can be shared to create faceted objects.

The PodGeometry Structure

A PodGeometry structure is a C-language struct that is defined as follows:

```
typedef struct PodGeometry
{
    float  fxmin, fymin, fzmin;
    float  fxextent, fyextent, fzextent;
    float  *pvertex;
    short  *pshared;
    uint16 vertexcount;
    uint16 sharedcount;
    short  *pindex;
    float  *puv;
} PodGeometry;
```

The following PodGeometry structure is used in the example *helloworld* program:

```
static PodGeometry gCornerGeometry = {
    /* float fxmin, fymin, fzmin */ -100.0, -100.0, -100.0,
    /* float fxextent, fyextent, fzextent */ 200.0, 200.0,
                                           200.0,
    /* float *pvertex */      gCornerVertices,
    /* short *pshared */      gCornerSharedVerts,
    /* uint16 vertexcount */ sizeof(gCornerVertices)/
                             kSizePerVertex,
    /* uint16 sharedcount */ sizeof(gCornerSharedVerts)/
                             (2 * sizeof(short)),
    /* short *pindex */      gCornerIndices,
    /* float *puv */         NULL
};
```

Example: Using Mercury Geometry to Construct an Object

Example 1-2 shows how the PodGeometry structure can be used in a Mercury application. The example constructs a faceted cube using Mercury geometry. It demonstrates the use of vertices, indices, (*u*, *v*) coordinates, and a PodGeometry header.

Example 1-2 Using Mercury Geometry

```
static float facetcubevtx[] =
{
    0,0,0, -1,0,0,      /* 0-x */
    0,0,100,0,-1,0,     /* 1-y */
    0,100,0,0,0,-1,     /* 2-z */
    0,100,100,0,1,0,    /* 3+y */
    100,0,0,1,0,0,      /* 4+x */
    100,0,100,0,0,1,    /* 5+z */
    100,100,0,1,0,0,    /* 6+x */
    100,100,100,1,0,0,  /* 7+x */
};

static short facetcubeshared[] =
```

```

{
    0,1,      /* 8-y */
    0,2,      /* 9-z */
    1,0,      /* 10-x */
    1,5,      /* 11+z */
    2,0,      /* 12-x */
    2,3,      /* 13+y */
    3,0,      /* 14-x */
    3,5,      /* 15+z */
    4,1,      /* 16-y */
    4,2,      /* 17-z */
    5,4,      /* 18+x */
    5,1,      /* 19-y */
    6,3,      /* 20+y */
    6,2,      /* 21-z */
    7,3,      /* 22+y */
    7,5,      /* 23+z */
};

static short facetcubeindex[] =
{
    0 + STXT + PCLK,2,10,12,14,      /* -x */
    8 + NEWS + CFAN + PCLK,16,1,19, /* -y */
    9 + NEWS + CFAN + PCLK,2,17,21, /* -z */
    13 + NEWS + CFAN + PCLK,3,20,22, /* +y */
    6 + NEWS + CFAN + PCLK,7,4,18,  /* +x */
    11 + NEWS + CFAN + PCLK,5,15,23, /* +z */
    0 + STXT + NEWS + CFAN + PCLK,-1,
};

static float facetcubeuv[] =
{
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
};

PodGeometry facetcube =
{
    0,0,0,
    100,100,100,
    facetcubevtx,
    facetcubeshared,
    sizeof(facetcubevtx)/(6*sizeof(float)),
    sizeof(facetcubeshared)/(2*sizeof(short)),
    facetcubeindex,
    facetcubeuv
};

```

Parts of a PodGeometry Structure

A PodGeometry object is divided into five parts:

- ◆ A 44-byte header
- ◆ A set of vertices
- ◆ A set of shared vertices
- ◆ Vertex indices for the object's strip fans (one `short` integer per index)
- ◆ The corresponding texture-map (u, v) coordinates (two `float` values for each u, v pair)

Each of these four parts of a PodGeometry object is described individually under the headings that follow.

The PodGeometry Header

The fields in a PodGeometry header are defined as follows:

Field	Contents
<code>xmin, ymin, zmin</code>	The minimum x, y , and z coordinates used in the object.
<code>xextent, yextent, zextent</code>	The maximum x, y , and z values used in the object, minus the minimum values used.
<code>pvertex</code>	Points to the array of vertices used. A vertex is an array of floats containing the coordinates of the vertex and either a normal vector (for lit cases) or a color (for pre-lit cases).
<code>pshared</code>	Points to an array of shared vertices
<code>vertexcount</code>	The number of vertices in <code>pvertex</code>
<code>sharedcount</code>	The number of shared vertices in <code>pvertex</code>
<code>pindex</code>	The array of indices.
<code>puv</code>	The array of (u, v) coordinates there is one (u, v) coordinate for each index in <code>pindex</code> .

The PodGeometry Vertices and Shared Vertices

The list of vertices in a PodGeometry structure is divided into two parts: a list of the actual vertices, followed by a list of `short` integers that point to the vertex and color information that is to be copied. In a PodGeometry structure, a vertex is made up of six floating-point numbers: specifically, x, y , and z followed by either nx, ny , and nz or r, g , and b . A shared vertex is made up of two short integers: a vertex index followed by a color index.

This architecture permits rapid execution when used with faceted geometry. As mentioned previously, Mercury vertices are transformed and lighted two at a time, so a dummy vertex must be added if there is an odd number of vertices.

Shared vertices are copied one at a time, so there is no need to pad your vertices out to an even number.

Faceted geometry uses only one normal for an entire face of an object. Consequently, color calculations can be reduced by calculating the color for only one of the points being used, and then copying that color along with vertex information for subsequent instances of the same normal. For example, if your application displays a faceted cube that has eight vertices and six normals, you

would normally have to transform and light 24 vertices to display the cube. By using Mercury's shared mechanism, you can light only eight vertices, and then make 16 copies. Of course, this solution yields correct lighting only with directional light, but in most cases it really doesn't look bad with the other kinds of lighting; try it and see.

The PodGeometry Indices

Two kinds of half-words used to describe pod geometry indices: vertex indices and texture numbers. Should a new texture need to be selected for a strip fan (which is always the case with the first strip fan in the object), the first vertex index has a magic bit set, and is followed by the number of the texture to select (a small integer). Requesting a load of texture -1 terminates the object.

There are five bits per index, so this format supports up to 2,048 vertices in a model:

```
#define PCLK0x8000
#define PFAN0x4000
#define CFAN0x2000
#define NEWS0x1000
#define STXT0x0800
```

Later in this section, each of the flags shown in this code fragment is described under a separate heading. Before we examine each individual flag, however, it might be helpful to take a look at how triangles are drawn in Mercury-based programs. That's because many of the flags used in the PodGeometry structures are used to determine how triangles are drawn in strips and fans. When you understand how triangles are drawn in a Mercury-based application, you'll have a better understanding of the flags that are provided in Mercury's PodGeometry structure.

The next subsection, "Drawing Clockwise and Counter-Clockwise Triangles," explains how triangles are drawn in Mercury programs.

Drawing Clockwise and Counter-Clockwise Triangles

Mercury uses a very fast set of algorithms for drawing triangles in strips and fans. To take advantage of this optimization when you construct a model, it helps to know whether the triangles used to render the model are drawn clockwise or counter-clockwise. Making this determination involves one small trade-off: the techniques used for drawing clockwise and counter-clockwise triangles in Mercury are not quite as intuitive as some older, more traditional strategies that you may be familiar with. But once you understand the Mercury system, it is not very difficult to use, and the extra speed is worth the extra effort.

Briefly, here's how the system works: When you want to draw a triangle in a strip or a fan, you use a flag called `prevclockwiseFLAG` to determine whether the previous triangle was clockwise or counter-clockwise.

Drawing Triangles in a Strip

If you are drawing triangles in a strip, Figure 1-2 shows what happens when you draw your triangles in this fashion. In the strip shown in Figure 1-2, the triangle at the left—which we will call Triangle 1 because it appears beneath the number 1—is the triangle that is drawn first. Notice that Triangle 1 is drawn in a clockwise

direction. The second triangle—which we will call Triangle 2 because it appears above the number 2—is drawn counter-clockwise. The third triangle, or Triangle 3, is drawn clockwise, and so on.

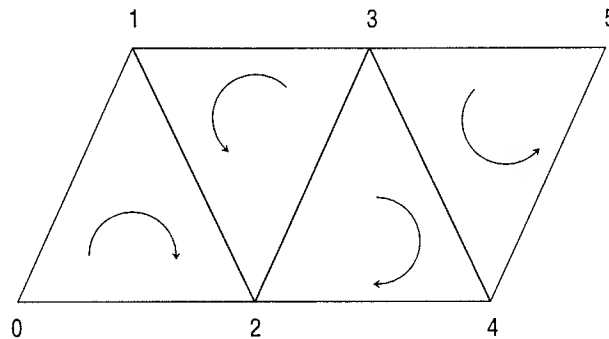


Figure 1-2 Drawing triangles in a strip.

When you are drawing triangles in a strip, another rule to remember is that each time you add a triangle, you replace the oldest vertex in the strip. This process continues until a complete strip is drawn. (When you draw triangles in a fan configuration rather than in a strip, the process is slightly different, as you'll see momentarily.)

Table 1-3 shows more details about how triangles are drawn in a strip using the "replace-oldest-vertex" rule.

Table 1-3 Using the "Replace-Oldest-Vertex" Rule

Slot	Triangle 1	Triangle 2	Triangle 3	Triangle 4
A	0	3	3	3
B	1	1	4	4
C	2	2	2	5
CW or CCW^a	CW	CCW	CW	CCW

a. Clockwise or counter-clockwise

Table 1-3 is just another mechanism for showing how the triangles illustrated in Figure 1-2 are drawn. Notice that the left-most column of Table 1-3 is labeled *Slot*. In Mercury, the word *slot* is used to describe a kind of container in which a number can be placed—in much the same way that a value can be placed in a register in the world of computer hardware.

When you are determining whether rotation is clockwise or counter-clockwise, follow the path from the vertex in Slot A to the vertex in Slot B to the vertex in Slot C.

For example, Table 1-3 shows how a group of three slots—named A, B, and C—can be used to hold a set of numbers that represent the three vertices of a triangle. This is a convenient mechanism for showing how the vertices of new triangles replace vertices of older triangles as new triangles are added to a strip. When you

draw a triangle, you can specify the order in which its vertices are drawn by a placing the vertex number into the appropriate slot. Then, each time a vertex of a new triangle replaces a vertex of an older triangle, you can specify which vertex of the older triangle is replaced by simply removing the old vertex from its slot and replacing it with the new one.

For example, refer to Triangle 1 in Table 1-3. When that triangle is drawn, Vertex 0 is placed in Slot A, Vertex 1 is placed in Slot B, and Vertex 2 is placed in Slot C.

Whether a triangle is drawn clockwise or counter-clockwise is determined by looking at the direction (clockwise or counter-clockwise) of the three vertices in Slot A, B, and C, in that order.

When it is time to draw Triangle 2, it is drawn in a counter-clockwise direction. Its vertices are numbered 1, 2, and 3, and Vertex 3 in Triangle 2 replaces Vertex 0 in Triangle 1, as shown in the table. This process continues as you add more triangles to the strip shown in Figure 1-2. When you draw Triangle 3, Vertex 4 replaces Vertex 1. When you draw Triangle 4, Vertex 5 replaces Vertex 2—and so on.

Drawing Triangles in a Fan

There is only one difference between drawing a triangle fan in Mercury and drawing a triangle strip. (Whether group of triangles is drawn in a strip or in a fan is determined by the `pfan` and `cfan` flags, as you'll see later in this section.) When you add a triangle to a strip, you don't replace the oldest vertex, as you do when you add a triangle to a strip. Instead, you replace the middle vertex of the most recently drawn triangles. This difference is illustrated in Figure 1-3.

Table 1-4 on Page 20 shows how the “replace-middle-vertex” rule works when you add a triangle to a fan.

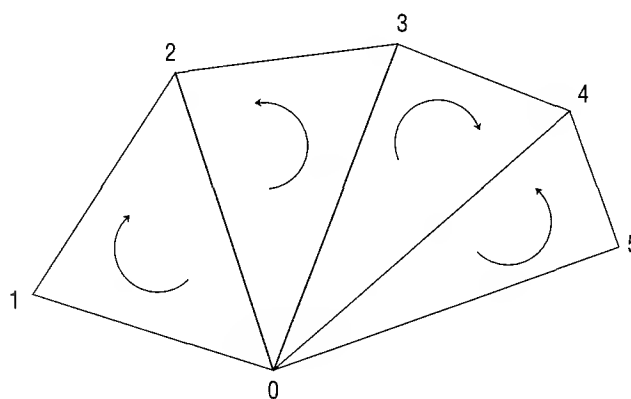


Figure 1-3 Adding triangles to a fan.

Table 1-4 Using the “Replace-Middle-Vertex” Rule

Slot	Triangle 1	Triangle 2	Triangle 3	Triangle 4
A	0	0	0	0
B	1	3	3	5
C	2	2	4	4
CW or CCW^a	CW	CCW	CW	CCW

a. Clockwise or counter-clockwise

To see what happens when you add a triangle to a fan, refer to Triangle 1 in Table 1-4. That triangle is drawn in the same way it was drawn earlier, when we added a triangle to a strip. Vertex 0 is placed in Slot A, Vertex 1 is placed in Slot B, and Vertex 2 is placed in Slot C.

When it is time to draw Triangle 2, however, the rules change. This time, Vertex 3 replaces Vertex 1 in Triangle 1. When you draw Triangle 3, Vertex 4 replaces Vertex 2 in Triangle 2. When you draw Triangle 4, Vertex 5 replaces Vertex 3—and so on.

Flags Used in the PodGeometry Structure

Now that you know how triangles are drawn in Mercury applications, it's time to examine the flags that are provided in Mercury's PodGeometry structure. In the paragraphs that follow, each flag used in the PodGeometry structure is described under a separate heading.

The PCLK Flag

The `prevclockwiseFLAG` is used if the previous index had a clockwise orientation. In both strips and fans, the orientation of triangles alternates between clockwise and counter clockwise, as shown in Figure 1-2 and Figure 1-3.

The PFAN and CFAN Flags

You should use these flags if the previous or current index has a “fan” type orientation (which means that the middle point of the previous triangle should be eliminated.)

Note: Because of the way in which the branching order works in the Mercury pipeline, the `curfanFLAG` must be set whenever the `startnewstripFLAG` is set.

The NEWS Flag

Use this flag if your index is the start of a new strip fan. Use of this on the first index in the object is optional.

The STXT Flag

You can use this flag if a different texture is needed for a strip fan. The next half-word is the texture number. The first strip fan in an object always needs a texture.

The PodGeometry (u, v) Coordinates

In a PodGeometry object, the (u, v) coordinates are normalized floating-point values. The value 0.0 is the smallest texture coordinate, and the value 1.0 is the largest. The normalized coordinate is scaled at run time based on the size of the biggest LOD of the texture. When a texture within a texture page is selected the draw routine also loads the *uscale* and *vscale* of the texture. If tiling is being used, the coordinate can be greater than 1.0.

The Pod Structure

The Pod structure is the highest-level data structure in Mercury. It associates a POD's geometry with a Matrix, a light list, a surface Material, one or more textures, as well as with the code which Mercury will call to actually render the object.

Benefits that Mercury derives from using Pod structures include the following:

- ◆ The user can easily change how an object looks by choosing different draw routines, textures, material properties, and lighting.
- ◆ A pointer to user data is also provided in the Pod structure, to allow custom lighting or rendering code to access user-specifiable information.
- ◆ The Pod structure contains a pointer to the next POD, permitting PODs to be grouped into linked lists.

The following Pod structure is used in the *helloworld* sample program:

```
Pod gCornerPod = {
    /* uint32 flags */                0,
    /* struct Pod *pNext */           NULL,
    /* void (*pcase)(CloseData*) */   M_SetupPreLit,
    /* struct PodTexture *ptexture */ NULL,
    /* struct PodGeometry *pgeometry */&gCornerGeometry,
    /* Matrix *pmatrix */              &gCornerMatrix,
    /* uint32 *plights */              gCornerLightList,
    /* uint32 *puserdata */            NULL,
    /* Material *pmaterial */          &gCornerMaterial
};
```

Declaration of the Pod Structure

The declaration of the Pod structure is defined as follows:

```
typedef struct Pod
{
    uint32 flags;
    struct Pod *pNext;
    void (*pcase)(CloseData*);
    struct PodTexture *ptexture;
    struct PodGeometry *pgeometry;
    Matrix *pmatrix;
    uint32 *plights;
    uint32 *puserdata;
    Material *pmaterial;
}
```

Flags Used in the Pod Structure

In most cases, the Mercury user sets the `flags` field of a Pod structure to 0. Then, when the `M_Sort` function is called, it optimizes the rendering of the objects by setting other flags. The `flags` field in the Pod structure is currently used for the following attributes:

```
#define samecaseFLAG(1 << (31-8))      /* the 0x00800000 bit */
#define sametextureFLAG(1 << (31-9))   /* the 0x00400000 bit */
#define callatstartFLAG(1 << (31-16)) /* the 0x00008000 bit */
#define casecodeisasmFLAG(1 << (31-17)) /* the 0x00004000 bit */
#define usercheckedclipFLAG(1 << (31-18)) /* the 0x00002000 bit */
#define specularFLAG(1 << (31-22))    /* the 0x00000200 bit */
#define hithernocullFLAG(1 << (31-23)) /* the 0x00000100 bit */
#define nocullFLAG(1 << (31-28))      /* the 0x00000008 bit */
#define frontcullFLAG(1 << (31-29))   /* the 0x00000004 bit */
#define clipFLAG(1 << (31-30))        /* the 0x00000002 bit */
#define callatendFLAG(1 << (31-31))   /* the 0x00000001 bit */
```

Each flag is described individually in the following paragraphs.

samecaseFLAG

Set this flag if a POD calls the same draw routine (using the `pod->pcase` field) as the previous POD. If this flag is set, the initialization routine for the draw routine does not have to be called (because it was called for the previous POD). The `M_Sort` function sets this flag automatically.

sametextureFLAG

Set this flag if a POD uses the same texture page as the previous POD. `M_Sort` sets this flag automatically, to minimize the number of texture loads at render time.

callatstartFLAG

If this flag is set, the routine pointed to by `puserdata` executes before calling the `M_Draw...` function.

casecodeisasmFLAG

If this flag is set, the `M_Draw` function does not save its registers off before calling `pod->pcase`. If this flag is not set, integer and floating-point registers are saved in RAM, and the `pcase` routine is called using C calling conventions (in particular, GPR3 contains a pointer to `CloseData`.)

usercheckedclipFLAG

If this flag is set, `M_Draw` does not check the object to make sure it is within the view volume. The POD being processed should be checked by user code to ensure that it does not clip with the left or top edges of the screen or the hither plane. Otherwise, the results are unpredictable.

specularFLAG

If this flag is set the camera position is inverse transformed into local object coordinates. This flag must be set for specular lighting to work properly.

hithernocullFLAG

If this flag is set and the object's bounding box is projecting through the hither plane, culling is disabled. Result: The Triangle Engine draws all triangles, whether back-facing or front-facing.

nocullFLAG

This flag disables backface culling for the affected POD. Result: The Triangle Engine draws all triangles, whether back-facing or front-facing.

frontcullFLAG

Normally, backface culling is enabled. If this flag is set, frontface culling is enabled instead.

clipFLAG

This flag is set by the bounding-box check in `M_Draw`. It signals to the rendering routine that clip-testing is enabled. If the `usercheckedclipFLAG` is set, this flag must also be set by the user.

callatendFLAG

If this flag is set, the routine pointed to by `puserdata` runs after calling the `M_Draw...` function.

Next-Pod Pointer (the `pod->pnext` field)

The `pod->pnext` field typically points to a "next pod to be rendered." `M_Sort` quickly arranges the linked list of PODs by changing only this field. Note that when calling `M_Sort`, a pointer to NULL does not signify an end-of-list situation. Rather, the user must specify a list length when calling `M_Sort`.

Setup Case (the `pod->pcase` field)

The `pod->pcase` field is a pointer to the function to be called to render the affected object. `M_Draw` calls this routine for the first object that needs this case (see the `samecaseFLAG`, described previously). Typically, this routine is a setup function such as `M_SetupPreLit` or `M_SetupDynLitTex`.

Pod Texture (the `pod->ptexture` field)

The `pod->ptexture` field is a pointer to the `PodTexture` structure describing the texture to be used for the affected POD. For untextured objects, this field is ignored.

Pod Geometry (the `pod->pgeometry` field)

The `pod->pgeometry` field is a pointer to a `PodGeometry` structure. This structure contains information about the vertices, texture coordinates, normals, and the alike.

Matrix (the `pod->pmatrix` field)

The `pod->pmatrix` field points to a `Matrix` data structure. A `Matrix` structure is a structure containing the local transform for a given object. A large matrix library is provided to perform operations for orientating and positioning an object by manipulating the affected matrix.

Light List (the pod->plights field)

The pod->plights field points to a list of lights to be applied to the object. The list, which is actually just an array of unsigned 32-bit integers, can be shared between multiple PODs that need similar lighting applied to them. The light list is described in the section where Lighting data structures are described.

User Data (the pod->puserdata field)

The pod->puserdata field points to an assembler function that is called if either the callatstartFLAG or callatendFLAG is set.

Material (the pod->pmaterial field)

The pod->pmaterial field points to a Material data structure. (The Material data structure is described in “The Material Structure” on page 26.)

Mercury Lighting

Mercury's lighting functions are multi-purpose procedures; along with performing lighting operations, Mercury's light functions can also manage specific per-vertex functions, such as creating fog.

The lighting used with a POD must match the requirements of the Draw function that is used with the pod, and must also match the texture blender and destination blender initialization operations used for the POD's case.

Mercury's Light Functions

The light functions available in Mercury are:

- ◆ M_LightFog
- ◆ M_LightDir
- ◆ M_LightPoint
- ◆ M_LightSoftSpot
- ◆ M_LightFogTrans
- ◆ M_LightDirSpec
- ◆ M_LightDirSpecTex

Mercury's light functions work a little differently from the other kinds of functions implemented in the Mercury API. In a Mercury-based game, you don't call Mercury's lighting routines directly. Instead, you construct a list of lights that are visible to each object or group of objects you want to illuminate. In this list, you associate each lighting function that you want to call with the POD object that you want to illuminate. Then, when you call the M_Draw function to draw each object that is to be illuminated, Mercury automatically provides the kind of lighting that you have specified.

Along with constructing a light list, you supply Mercury with two other kinds of data structures for each POD you want to light: a *lighting structure* that describes the attributes of the light you want to create for the POD, and a *Material structure* that describes the kind of material that is being illuminated. When you have provided these two structures, Mercury has all it needs to call the appropriate lighting functions.

Light Lists

As noted in the previous section, every application that uses Mercury's lighting functions is equipped with a *light list*. A light list is a data structure that lists the lighting functions used in your application and some of their attributes. For example, in the *helloworld* example program, the following light list is defined in a file named *filepod.c*:

```
uint32 gModelLightList[] = {
    (uint32)&M_LightDir,
    (uint32)&gDir,
    (uint32)&M_LightPoint,
    (uint32)&gPoint,
    0
};
```

In Mercury, a light list is a list of pointers to light functions and data structures that are used by those functions. Every light function used in Mercury is associated with a specific kind of data structure that is used to define the characteristics of the lighting that the function provides. In a light list, each function that is listed is followed by a pointer to its corresponding data structure. The data structures used with Mercury's light functions are described in more detail under the next heading, "Lighting Structures and Material Structures."

When you have provided your application with a light list, your light list determines what kinds of operations take place when your application calls the `M_Draw` command.

Lighting Structures and Material Structures

Recall that each light function used in Mercury is associated with two data structures: a *lighting structure*, which describes the kind of lighting you want to use to illuminate a particular POD, and a *Material structure*, which sets the material properties of the POD that is being illuminated. Each lighting function defined in Mercury uses a different kind of lighting structure. But Mercury defines only one kind of Material structure, which is used by all lighting functions.

Lighting structures are described in more detail under the next heading, "How Lighting Structures Work." Material structures are examined in the subsection headed "The Material Structure" on page 26.

How Lighting Structures Work

Because different kinds of operations have different effects, each light function used in Mercury is associated with a different kind of data structure that must be filled in before its corresponding light function is called. For example, before the *helloworld* program calls the `M_LightDir` function, the program sets the fields of the data structure `LightDir` to the values shown in the following code fragment.

When you call the lighting function that uses the data structure you have filled in, the function computes the *r*, *g*, *b*, and *a* components (or as many of those components as needed) for the surface at the point you have specified.

The *helloworld* program uses two lighting structures—a `LightPoint` structure named `gPoint` and a `LightDir` structure named `gDir`—which are defined in the *filepod.c* file:

```
static LightPoint    gPoint = {
    /* float x, y, z */      -60.0, 60.0, 60.0,
    /* float maxdist */     25000.0 * 256.0 * 0.9,
    /* float intensity */   25000.0,
    /* Color3 lightColor */ { 0.7, 0.7, 0.9 }
};

static LightDir      gDir = {
    /* float nx, ny, nz */   0.3, -0.7, 0.648,
    /* Color3 lightcolor */  0.2, 0.2, 0.2
};
```

Pointers to structures `gPoint` and `gDir` appear in the light list presented earlier in the section, under the heading “Light Lists” on page 25.

The *helloworld* program associates each of these structures with a POD in the following `for`, which appears in the *filepod.c* source file:

```
*maxPodVerts = 0;
curPod = gBSDF->pods;

for (i=0; i<gBSDF->numPods; i++) {
    if (curPod->pgeometry->vertexcount > *maxPodVerts)
        *maxPodVerts = curPod->pgeometry->vertexcount;
    Model_Scale(curPod, scaleFactor);
    curPod->plights = gModelLightList;
    curPod = curPod->pnext;
}
```

Once this code is executed, the specified kind of lighting is created each time `M_Draw` is called to draw one of PODs shown in the preceding example.

The Material Structure

Along with a lighting structure that defines the kind of lighting being created, each Mercury lighting function is also associated with a `Material` structure that defines the characteristics of the object being lighted.

Each POD object used in Mercury is equipped with a pointer named `pmaterial`, which points to a `Material` structure. In Mercury, a `Material` structure is defined as follows:

```
typedef struct Material
{
    Color4 base;
    Color3 diffuse;
    float shine;
    Color3 specular;
} Material
```

Mercury's `Material` structure is a little different from materials structures used in some other APIs. The main difference is that in Mercury's `Material` struct, the ambient and emissive values have been combined into the base color. The formula to combine the colors is:

$$Ma * La + Me$$

where:

- ◆ *Ma* is the ambient material property
- ◆ *La* is the ambient light in the scene
- ◆ *Me* is the emissive material property.

Since this only needs to be calculated when the ambient light in the scene changes, Mercury combines them to reduce the number of per-object calculations.

If the object is transparent, the user puts the alpha base value in the base color.

The `diffuse` color is multiplied by the value of each light, and a `diffuse` color for each light used is stored in a buffer named `convlightdata`.

The `shine` value is the specular exponent for the object. It defines how wide the highlight is.

The specular color is multiplied by the value of each specular light, and a specular color for each light used is stored in the `convlightdata` buffer for each specular light used.

For each object being lighted, a base color and a `diffuse` color are copied into `closedata`. If the `specularFLAG` is set the `shine` and specular color are also copied into `closedata`. In addition, the camera's position is inverse-transformed into the object's local coordinates.

Using Lighting Structures and Material Structures

To specify a lighting function and the pointer to its lighting structure, set the `plights` field in the `POD` structure. Similarly, set the `pmaterial` field in the `POD` to specify the `Material` structure.

The `plights` field points to a zero-terminated list of pointer tuples. The first member of the tuple is an address of the light routine. The second tuple member is a pointer to the data that represents the light. To call a lighting function, first initialize the data that the call requires, and then make the call.

The Calling Sequence of Lighting Operations

One important characteristic of Mercury's lighting routines is that they are always called in a specific sequence. The first time you call a Mercury lighting routine, its input is the `rgb` code of the base color, and its `u` value is 1.0. When the first lighting routine returns, its output—which is returned in the form `r, g, b, a`—is used as the input for the next lighting routine that is called. This process continues until all lighting routines have been called.

More details about the parameters, return values, and other attributes used with Mercury's light functions see Appendix A, "Parts of the Mercury Engine."

Varieties of Lighting Used in Mercury

In addition to the object's color, the lighting code in Mercury can compute three different kinds of lighting effects: fog, transparency with fog, and specular lighting.

Fog Lighting

Mercury's lighting code computes an alpha value that is passed unchanged through the texture blender. Then the alpha value goes to the destination blender, where it is blended with the fog color in a constant register using the following formula:

$$(\text{alpha} * \text{texture blender color}) + (\text{fog color} * (1 - \text{alpha}))$$

Transparency with Fog

The lighting code also computes a `UCoordColor` value that is used to look up an intensity value from a special texture map that is 17 entries long. `UCoordColor=0` maps to 0, and `UCoordColor=16` maps to `0xff`. The texture intensity is blended in the texture blender with the fog color in a constant register using the formula:

$$(\text{UCoordColor} * \text{color}) + (\text{fog color} * (1 - \text{UCoordColor}))$$

The final color is then destination blended with the frame buffer using the vertex alpha value for the transparency effect.

Specular Lighting

To create specular lighting, the alpha value computed by Mercury's lighting code is passed unchanged through the texture blender. Then it also goes to the destination blender, where it is multiplied by the specular color in a constant register and added to the output of the texture blender color using the following formula:

$$\text{texture blender color} + (\text{alpha} * \text{specular color})$$

Appendix A, "Parts of the Mercury Engine," explains how each of these varieties of lighting works when you call each of Mercury's lighting functions.

Note: *Mercury's lighting code cannot compute transparency with texture and specular lighting in a single pass because that kind of operation requires the use of the destination blender for two different operations: specular and transparency.*

The Two Stages of a Mercury Lighting Routine

Mercury executes its lighting routines in two distinct stages: an *initialization* stage and a *per vertex* stage.

The Initialization Stage

In the initialization stage of lighting operations, your application calls routines to initialize all the parameters that are required by each lighting function in the light list. These operations include multiplying the diffuse and specular colors by the value of the light color, and then transforming the light's direction and position into the POD's local coordinates.

The Per Vertex Stage

In the per vertex stage, the colors for the vertex are calculated. If fog is used, it is also included in the light routines executed at this stage of processing. If fog is calculated, the result of the calculation is returned either in the alpha channel or (in the case of transparency with fog) in the *u* texture coordinate.

Appendix A, "Parts of the Mercury Engine," explains how the initialization and per-vertex stages work in each of Mercury's lighting routines.

Example: Using Lighting Data

The following code fragment is an example of light data that might be used to represent a POD that has fog, one directional light source, and one point light source.

Example 1-3 *Lighting Data*

```
LightFog LightFogData =
{
    1.0/(30.0 - 10.0),10.0/(30.0 - 10.0)
};

LightDir LightDirData =
{
    -0.866,0.0,-0.5,
    0.25,0.25,0.25,
};

LightPoint LightPointData =
{
    -0.866,0.0,-0.5,
    250000.0 * 32.0 * 1.0,
    250000.0,
    1.0,1.0,1.0,
};

long planelights[] =
{
    (long)&M_LightFog,
    (long)&LightFogData,
    (long)&M_LightDir,
    (long)&LightDirData,
    (long)&M_LightPoint,
    (long)&LightPointData,
    0
};
```

Camera Operations in Mercury

One of the first steps in developing a Mercury-based application is to create and set up a *camera skew matrix*. A camera skew matrix is a specially formatted matrix that is stored in Mercury's `CloseData` structure and is used to transform vertices into screen coordinates.

To construct the camera skew matrix for Mercury, two special base matrices are composited together using the equation

$$F = CS$$

where F is a camera skew matrix, C is a camera coordinate transformation, and S is a source skew matrix .

The camera coordinate transform (C) transforms the world coordinate system in such a way that the objects in the scene are down the negative z axis from the origin. The skew matrix (S) scales into the screen coordinates and provides the desired perspective.

To simplify the process of creating and maintaining the camera skew matrix, Mercury provides a pair of utility functions named `M_SetCamera` and `Matrix_Perspective`. These two functions are described in more detail under the headings that follow.

For an example that shows how a camera skew matrix can be set up in a Mercury program, see the subsection under the heading "Creating and Setting Up a Camera" on page 41.

The `M_SetCamera` Function

`M_SetCamera` takes a normal camera matrix in world space and multiplies it with a specially formatted skew matrix to form Mercury's internal camera skew matrix.

The syntax of the `M_SetCamera` function is:

```
void M_SetCamera( CloseData *close, Matrix *normalCameraMatrix,
                  Matrix *specialSkewMatrix )
```

Arguments expected by `M_SetCamera` are:

<code>close</code>	a pointer to a <code>CloseData</code> structure
<code>NormalCameraMatrix</code>	a pointer to a matrix that describes the camera's position in world space
<code>SpecialSkewMatrix</code>	a pointer to a specially formatted matrix that contains perspective and screen projection information. You can create this matrix by calling <code>Matrix_Perspective</code> , described under the next heading.

The `Matrix_Perspective` Function

The `Matrix_Perspective` function creates a specially formatted skew matrix using a basic description of the frame buffer geometry and viewing frustum.

The syntax of the `Matrix_Perspective` function is,

```
void Matrix_Perspectiv( Matrix *specialSkewMatrix, ViewPyramid
                        *view, float screenXMin, float screenXMax, float screenYMin,
                        float screenYMax, float scaleW )
```

Arguments expected by `Matrix_Perspective` are:

<code>specialSkewMatrix</code>	a pointer to a Matrix that is used to return the calculated matrix
<code>View</code>	a pointer to a view pyramid. (The view pyramid describes the basic view frustum; (see the <i>vp.</i> calls in Example 2-3 on Page 41.)
<code>ScreenXMin</code>	the minimum screen X position
<code>ScreenXMax</code>	the maximum screen X position
<code>ScreenYMin</code>	the minimum screen Y position
<code>ScreenYMax</code>	the maximum screen Y position
<code>ScaleW</code>	a scaler for <i>w</i> .

How a Skew Matrix Works

In Mercury, a skew matrix is implemented as an object called a `Matrix` structure. a `Matrix` structure is simply a code representation of a skew matrix, which can be represented as follows in a Mercury program:

```
Matrix camMtx = {
{
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    0.0, 0.0, 0.0
}
```

Each matrix created using the $F = CS$ equation is a 4-by-4 matrix. Internally, however, Mercury represents each such matrix as a 3-by-4 matrix. To reduce a 4-by-4 matrix to a 3-by-4 matrix, you simply drop a column. Usually, the dropped column is the fourth column, which corresponds to the homogeneous coordinate *w*. However, in perspective operations, the third column, representing *z*, is dropped instead, and the homogeneous coordinate is kept in order to perform the perspective.

When a point is transformed in world coordinates $(x, y, z, 1)$ by *F* (the camera skew matrix in the preceding equation), the result is a screen coordinate (x, y) ready for the M2 triangle engine and a *w* value the reciprocal of which is sent to the triangle engine.

A Camera Matrix in World Space

A camera matrix in world space is created like any other transformation in world space. For example, in the *helloworld* program, the following code positions and orients the camera:

```
Matrix_SetTranslationByVector(&camMtx, camLocation );
Vector3D_Zero( &lookAtPt );

Matrix_LookAt(&camMtx, &lookAtPt, camLocation, 0.0);
```

In this code fragment, the `Matrix_SetTranslationByVector` function positions the camera at `camLocation`, and the `Matrix_LookAt` function orients the camera to look at the origin (0, 0, 0). (For more details on the operation of these functions, see the appendices in this book and your Mercury online documentation).

Constructing the Skew Matrix

The output of applying the camera transformation are coordinates (x', y', z') which are oriented along the screen but need to be scaled to the resolution of the screen. In addition, the origin of the screen is the upper left hand corner and the screen's y axis points downward. The skew matrix S must be constructed by concatenating a perspective transform, P , with a scale and translate transformation T that produces screen coordinates.

$$S = PT$$

In this equation, the perspective transformation, P , provides the foreshortening of the scene in camera coordinates. It can be specified by the relationship between how far the camera is from the projection plane relative to the size of the projection plane. Figure 1-4 shows how such a perspective transformation can be built in terms of a view frustum defined by the following quantities:

- ◆ t —the position of the center point relative to the top border of the projection plane
- ◆ b —the position of the center point relative to the bottom border of the projection plane
- ◆ r —the position of the center point relative to the right border of the projection plane
- ◆ l —the position of the center point relative to the left border of the projection plane
- ◆ n —the position of the projection plane from the eye point
- ◆ x_{\max}, y_{\max} —the pixel coordinates of the bottom right hand corner of the screen,
- ◆ x_{\min}, y_{\min} —the pixel coordinates of the top left hand corner of the screen.

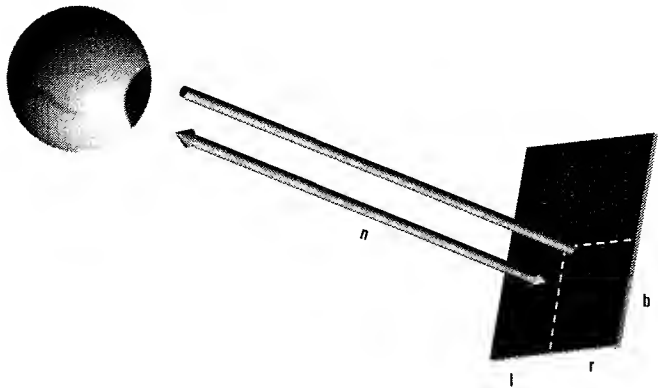


Figure 1-4 Building a perspective transformation.

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \text{XX} & 0 \\ 0 & \frac{2n}{t-b} & \text{XX} & 0 \\ \frac{r+1}{r-l} & \frac{t+b}{t-b} & \text{XX} & -1 \\ 0 & 0 & \text{XX} & 0 \end{bmatrix}$$

$$T = \begin{bmatrix} \frac{x_{\max} - x_{\min}}{2} & 0 & 0 & 0 \\ 0 & \frac{y_{\max} - y_{\min}}{2} & 0 & 0 \\ \text{XX} & \text{XX} & \text{XX} & \text{XX} \\ \frac{x_{\max} + x_{\min}}{2} & \frac{y_{\max} + y_{\min}}{2} & 0 & 1 \end{bmatrix}$$

Figure 1-5 Illustrations of skew matrices.

Figure 1-5 shows how the helper function `Matrix_Perspective` builds the skew matrix when the view pyramid data and the screen dimensions are passed to it. This function is passed a value of `1.0/hither` to scale the w value so that $w = 1.0$ is at the hither plane.

In the illustration, the columns and rows that are deleted when the matrix is reduced from a 4-by-4 configuration to a 3-by-3 configuration (see “How a Skew Matrix Works” on page 31) appear against a gray background.

Mercury Texture-Mapping

As explained in the *3DO Command List Toolkit* manual, M2 stores textures in a 16K block of RAM called *texture RAM*, or TRAM. Because of this texture-mapping architecture, the most efficient way to process different objects that access the same 16K of texture information is to process them at the same time. By processing textured objects in this way, you can avoid reloads of the same data into TRAM.

Mercury works very efficiently with the texture-mapping architecture of M2 because it provides a 16K block of TRAM that can contain many textures that are used at the same time. Moreover, Mercury sorts PODs in such a way that when multiple PODs need the same texture, they are processed at the same time, making multiple loads of TRAM unnecessary.

For instance, the *bigcircle* example provided on the M2 distribution media loads the TRAM 30 different times in a frame with two different kinds of textures:

- ◆ A 16K page containing three 8-bit textures (measuring 64 by 64 texels), mipmapped to four levels of detail
- ◆ A 16K page containing twelve 8-bit textures (measuring 32 by 32 texels), mipmapped to four levels of detail.

For more information on M2 texture mapping, see the *3DO Command List Toolkit* manual.

Summary

This chapter introduced the components of the Mercury rendering engine, described their attributes, and provided some isolated examples showing how to use those components in an M2 application. In Chapter 2, "Using Mercury," you'll see how Mercury can be used to create an actual application.

Using Mercury

At the beginning of Chapter 1, “Introducing Mercury,” you were presented with a list of basic steps for creating and building a game using the Mercury rendering engine. This chapter contains the same list of steps, described in considerably more detail.

To demonstrate the process of creating a Mercury application, this chapter uses the same sample application that was used to illustrate important points in Chapter 1: the *helloworld* example program that is provided on your M2 distribution CD-ROM.

This chapter is divided into the following sections:

Topic	Page
Anatomy of a Mercury Application	36
About the helloworld Program	36
Preparing to Write a Mercury Game	37
Creating and Rendering Frames	42
Writing Your Game Code	43
Cleaning Up	45
Summary	46

Anatomy of a Mercury Application

As noted in the section titled “Creating an Application with Mercury” on page 3, the following general steps are typically followed to create a game using Mercury:

1. Perform the initialization operations your application requires. Initialization of Mercury is a three-stage operation that requires the following steps:
 - ◆ Create and initialize a GState object
 - ◆ Initializing the Mercury rendering engine
 - ◆ Setting up and initializing a camera.

Each of these operations is examined in more detail under a separate heading later in this chapter.

2. Perform the work that is needed to create and render each frame in your game. If you are using transparency, set the `srcaddr` field of your `CloseData` structure to the address of your current destination bitmap. Then write code that performs the standard per-frame operations, such as performing frame-buffering operations and page-flipping. This section of code also initializes the M2 destination blender.
3. Write the code that is required to run your game. In this section of code, you update all POD transforms and add new PODs to your game. Typically, at runtime, a game written using Mercury calls various Mercury and GState functions to perform such operations as:
 - ◆ Drawing models and backgrounds
 - ◆ Displaying rendered images
 - ◆ Updating camera positions as necessary
 - ◆ Checking bounding boxes of the objects used in your game
 - ◆ Determining what PODs should be displayed, and displaying them
 - ◆ Handling user input.
4. Write code that shuts your game down and performs all needed cleanup functions when the user terminates the program.

The remainder of this chapter uses the *helloworld* program to describe and illustrate the preceding steps in much more detail.

About the helloworld Program

The *helloworld* program is a bare-bones example of a Mercury application. It doesn't begin to demonstrate everything Mercury can do, but it is an excellent example of a basic framework that can be used as a starting point for creating a Mercury-based game.

Specifically, *helloworld* performs the following operations:

- ◆ Initializes a GState object, some bitmaps, and a display in preparation for rendering (in the *graphicsenv.c* file).
- ◆ Creates some simple geometry (in the *data.c* file) that is later used for two walls and a floor.
- ◆ Reads in a 3D model (in *filepod.c* and *bsdfreader.c*)
- ◆ Displays a single frame of data containing both a background POD and a model POD. The code that implements these operations appears in the program's main file, *helloworld.c*.

When you execute *helloworld*, you can position the camera anywhere you like using command-line arguments. Because the camera is aimed at the center of the input model's geometry, the scene that the program creates is always visible.

The program uses a standard texture reader (implemented in *tex_read.c*) and a standard SDF-file reader (implemented in *bsdf_read.c*). Both these readers can read data from the standard 3DO M2 graphics conversion tools.

When the user presses any control-pad button, the program terminates.

Preparing to Write a Mercury Game

The first step in creating an application with Mercury is to perform the initialization operations that the application requires. Typically, Mercury initialization operations include the following steps:

1. Using the GState (Graphics State) calls provided in the M2 Graphics Portfolio, create and initialize a GState object. (For more information about GState objects, see the *3DO Command List Toolkit* manual.)
2. Using other function calls provided in the M2 Graphics Portfolio, set up a graphics environment for your application. (In the *helloworld* program, you can find the code that sets up a graphics environment in a file named *graphicsenv.c*. You'll get a chance to take a closer look at that file later in this chapter.)
3. Organize a set of data that defines the geometry that your application requires. Typically, this collection of data provides information that describes your application's matrix, vertex, lighting, and texturing specifications, along with any other data that Mercury may require to handle the PODs used by your application. (The *helloworld* program makes use of a data file named *data.c*.)
4. Load and initialize the data that is needed to create and render the 3D models you will be using in your application. Typically, this data includes texture, lighting, boundary, and scaling information, along with any other data that

Mercury requires to create and render the models used in your application. (Model information used by the *helloworld* example application is provided in a file named *filepod.c*.)

5. Initialize a `CloseData` structure including elements that won't change through out the title. (In Mercury, a structure called the `CloseData` structure contains all input to the graphics pipeline. For details, see "The `CloseData` Structure" on page 12.) To initialize a `CloseData` structure, you call the `M_Init` function, which initializes Mercury, and the `M_PackColor` function, which sets the values of the colors used in your application. (For more details on these function, see this book's appendices and your on-line Mercury documentation.) In the *helloworld* program, `M_Init` is called in a file named *helloworld.c*. You'll see how the `M_Init` function works in the *helloworld* example program.
6. Set up and initialize a camera for your application by following these steps: First, call the `Matrix_Perspective` function to initialize a skew matrix that includes perspective. Then initialize your camera by calling the `M_SetCamera` function (another function that is called in the *helloworld* program's *helloworld.c* file.)
7. Perform the lighting functions that your application requires. Call the `M_Prelight` function to prelight all pods that will be lighting using a pre-lit case:

```
M_Prelight(ppodsrc, ppoddst, pclosedata);
```

Each operation in the preceding list is described under a separate heading in the paragraphs that follow.

Creating and Initializing a GState Object

When you start writing an application using Mercury, the first initialization operation you must perform is to create a GState object. Then you must set up and initialize a `CloseData` structure, and set up and initialize a camera.

In Mercury, the GState object—which is described in *3DO Command List Toolkit* manual—is a low-level object that all M2 libraries use to communicate with the M2 Triangle Engine. The GState object's main job is to encapsulate command list buffers using a format that all M2 libraries and folios can understand.

Mercury—like the Graphics Framework, the Graphics Pipeline, and all other M2 APIs—communicates with the Triangle Engine through the GState object. The GState objects provides Mercury with low-level routines that it can call to create command-list buffers and to send command-list buffers to the Triangle Engine for rendering into screen images.

Mercury uses the GState object to initialize the CloseData structure, a structure that contains all Mercury input to the graphics pipeline. (The CloseData structure is described in more detail in “The CloseData Structure” on page 12).

Creating a GState Object with the GS_Create Function

A Mercury-based application, like any other M2 program, initializes a GState object by calling the Graphics Portfolio function GS_Create. In the *helloworld* application, GS_Create is called in the sequence of code shown in Example 2-1.

Example 2-1 Creating and Initializing a GState Object

```
/* Create and set up the GState */
genv->gs = GS_Create();
if (genv->gs < 0) {
    retVal = Hg_NoMem;
    goto Failed_GSCreate;
}
err = GS_AllocLists(genv->gs, kNumCmdListBufs, kCmdListBufSize);
if (err < 0) {
    retVal = err;
    goto Failed_GSAllocLists;
}
GS_SetVidSignal(genv->gs, genv->d->signal);
GS_SetDestBuffer(genv->gs, genv->bitmaps[0]);

if (USE_Z_BUFFERING)
    GS_SetZBuffer(genv->gs, genv->bitmaps[genv->d->numScreens]);

/* Everything went OK */
retVal = 0;
printf("Using %d-bit display, size=%d x %d, %d bitmaps, %s\n",
genv->d->depth, genv->d->width, genv->d->height, genv->
    d->numScreens,
    (USE_Z_BUFFERING ? "Z-buffered" : "No Z-buffer"));
goto Exit;
```

The code shown in Example 2-1 appears in a file named *graphicsenv.c*, which also performs a number of other functions that are used to set up a graphics environment for the *helloworld* program.

Two of those functions, GS_SetVidSignal and GS_SetDestBuffer, appear in Example 2-1. If you examine the *graphicsenv.c* file, you'll see other functions that perform important operations such as allocating memory for graphics, creating and displaying a view, and loading and displaying bitmaps.

The *3DO Command List Toolkit* manual provides more detailed information about GState objects and how they are used in M2 programs.

Initializing the Mercury Rendering Engine

When your application has created and initialized a GState object, you're ready to initialize the Mercury rendering engine.

To initialize Mercury, you create and set up a structure called a `CloseData` structure. `CloseData` is a very important structure in Mercury because it contains all Mercury input to the graphics pipeline. `CloseData` is also used for temporary storage of intermediate calculations and temporary data.

To set up a `CloseData` structure, you call the Mercury function `M_Init`. (Appendix A, "Parts of the Mercury Engine," describes the syntax of the `M_Init` function.)

Initializing Mercury in the helloworld Program

Once a `CloseData` structure is set up, the application can set important fields in that structure by simply using the C assignment operator (`=`). The *helloworld* program sets up a `CloseData` structure in Example 2-2, which you can find in the *helloworld.c* file:

Example 2-2 Initializing Mercury

```
/* Initialize Mercury */
maxPodVerts = (maxPodVerts + 3) & ~3;
appData->close = M_Init(maxPodVerts, kMaxCmdListWords, genv->gs);
if (appData->close == NULL) {
    printf("Couldn't init Mercury.  Exiting\n");
    exit(1);
}
appData->close->fwclose = 1.01;
appData->close->fwfar = 100000.0;
appData->close->fscreenwidth = genv->d->width;
appData->close->fscreenheight = genv->d->height;
appData->close->depth = genv->d->depth;
```

In Example 2-2, the `CloseData` structure that is being set up is named `close`. It is referred to as `appData->close` because it is nested inside a larger structure named `appData`. In *helloworld*, `appData` a structure that contains both a `CloseData` structure and a `Matrix` structure. A `Matrix` structure—as you may recall from Chapter 1, "Introducing Mercury"—is a structure used to describe the location of a graphics object (such as a camera, pod, or the like).

The `CloseData` structure is described in more detail in “The `CloseData` Structure” on page 12. Under the next heading, “Creating and Setting Up a Camera,” you’ll learn more about how *helloworld* uses the `Matrix` structure.

Creating and Setting Up a Camera

The last step in the Mercury initialization process is to create and set up a camera. Set up and initialize a camera for your application by following these steps:

1. Create and initialize a special kind of matrix called a *camera skew matrix*—that is, a matrix that includes perspective.
2. Call the `Matrix_Perspective` function to initialize camera skew matrix—that is, a skew matrix that includes perspective.
3. Initialize your camera by calling the `M_SetCamera` function.

In Mercury, the first step in setting up a camera is to create a camera skew matrix (for details, see “Camera Operations in Mercury” on page 30).

A camera skew matrix can be constructed using a `Matrix` data structure. In Mercury, a `Matrix` structure is simply a code representation of a transformation matrix. For example, the camera skew matrix used in the *helloworld* program is defined as follows in the *helloworld.c* file:

```
Matrix camMtx = {
{
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    0.0, 0.0, 0.0
}
```

When you have created a camera skew matrix, you can set up a camera using a code sequence that looks something like the one shown in Example 2-3 (this code sequence is also taken from the *helloworld* program’s *helloworld.c* source file.)

Example 2-3 Creating and Setting Up a Camera

```
/* Set up the camera */

appData->skewMatrix = AllocMem(sizeof(Matrix), MEMTYPE_NORMAL);
if (appData->skewMatrix == NULL) {
    printf("Couldn't allocate skew matrix.  Exiting\n");
    exit(1);
}

vp.left = -1.0;
```

```
vp.right = 1.0;
vp.top = 0.75;    /* Basic 4/3 aspect ratio, with positive Y UP */
vp.bottom = -0.75;
vp.hither = 1.2;

Matrix_Perspective(appData->skewMatrix, &vp, 0.0,
    (float)(genv->d->width), 0.0, (float)(genv->d->height),
    1.0/gHitherPlane);

Matrix_SetTranslationByVector(&camMtx, camLocation );
Vector3D_Zero( &lookAtPt );

Matrix_LookAt(&camMtx, &lookAtPt, camLocation, 0.0);

M_SetCamera( appData->close, &camMtx, appData->skewMatrix );
```

The code shown in Example 2-3 allocates memory for a camera skew matrix, sets fields in a ViewPyramid object named `vp`, and then performs some perspective-related operations by calling a series of functions named `Matrix_Perspective`, `Matrix_SetTranslationByVector`, `Vector3D_Zero`, and `Matrix_LookAt`. Finally, the application's camera is initialized with the call to `M_SetCamera` shown in the last line of the listing. (For more information about the `M_SetCamera` function, see "The `M_SetCamera` Function" on page 30.)

To learn more about the ViewPyramid object and the other function calls shown in Example 2-3, refer to this book's appendices and your on-line M2 documentation.

Creating and Rendering Frames

When you have set up a camera for your application, you are ready to perform the work that is needed to create and render each frame in your title. At this stage of preprocessing, your application's game code calls various Mercury functions and Command List Toolkit (CLT) functions. to perform such operations as setting up the M2 destination blender, clearing the screen, and setting Z-buffers.

If you are using transparency, set the `srcaddr` field of your `CloseData` structure to the address of your current destination bitmap (`dstBM` in the following example):

```
pclosedata->srcaddr = dstBM->bm_Buffer;
```

Next, perform the usual per-frame operations, such as clearing the frame buffer and dispatching initialization commands. Typically, Mercury applications clear the frame buffer by calling the Command List Toolkit function

CLT_ClearFrameBuffer, and then initialize the M2 destination blender by calling the Mercury function M_DBInit:

```
CLT_ClearFrameBuffer(gs, .5, .5, .5, 1., 1, 1);
M_DBInit(GS_Ptr(gs), 0,0,320, 240);
```

In the *helloworld* program, the CLT_ClearFrameBuffer and M_DBInit functions are called in a function named Program_Begin, which you can find in the *helloworld.c* file.

Writing Your Game Code

When you've finished writing whatever frame-buffering code your title requires, you write your game code. In this section of your program, you update all POD transforms and add new PODs to your game.

Typically, your application's game code calls a few more Mercury and GState functions to perform such operations as:

- ◆ Drawing models and backgrounds
- ◆ Sending the command list to the Triangle Engine
- ◆ Checking bounding boxes of the objects used in your game
- ◆ Displaying PODs
- ◆ Handling user input.

Also, each time you update the position your application's camera, you must compute a camera matrix and combine it with a screen skew matrix:

```
pclosedata->fcamx = 0;
pclosedata->fcamy = -400;
pclosedata->fcamz = -600;
cameramat.Matrix_Data[i][j] = ...
Matrix_Mult((pMatrix)&pclosedata->fcamskewmatrix, &cameramat,
&skewmat);
```

Performing Bounds-Testing

Optionally, you can test the bounds of object groups to see which group of objects are visible:

```
M_BoundsTest(pbox, pclosedata);
```

After performing this test, you can check each bounding box and see if it has been found to be visible. If so, you must add any extra PODs that you need to the list of PODs in your `ppodbuffer`.

Displaying Your Game's PODs

Finally, you keep track of the PODs used in your game and display them as needed. In the *helloworld* program, this is the section of code that displays the program's model and background pods (named `firstPod` and `gCornerPod`, respectively):

```
/* Draw the model */
firstPod = M_Sort(modelPodCount, gBSDF->pods, NULL);
M_Draw(firstPod, app->close);

/* Now draw the background. We could have just appended them
   together above, but this is how to do it when you need to
   render multiple pod lists. */

M_Draw(&gCornerPod, app->close);
```

When that's all done, you can swap your frame buffers.

The Game-Play Section of the *helloworld* Program

In the *helloworld* program, all game code appears in a function named `Program_Begin`. (Yes, that's the same `Program_Begin` function that contains the `CLT_ClearFrameBuffer` and `M_DBInit` functions described in the preceding section, "Creating and Rendering Frames").

Example 2-4 on page 44 is a complete source-code listing of the complete `Program_Begin` function. Examine the comments in the listing carefully; they explain exactly how the action segment of the *helloworld* program works, all the way from setting up the M2 destination blender to responding to the user's control commands.

Example 2-4 The `Program_Begin` Function

```
void Program_Begin(GraphicsEnv* genv, AppData* app,
                   uint32 modelPodCount)
{
    ControlPadEventData cped;
    Pod                  *firstPod;

    /* First, set up DBlender so that the clear will also clear
       Z, etc. */

    GS_Reserve(genv->gs, M_DBInit_Size);
    M_DBInit(GS_Ptr(genv->gs), 0, 0, genv->d->width,
```



```
    genv->d->height);

/* Clear screen and Z-buffer */
CLT_ClearFrameBuffer(genv->gs, 0.9, 0.9, 0.92, 1.0, TRUE, TRUE);

/* Now reset state, since CLT_ClearFB changed the state of
   the DBlender */

GS_Reserve(genv->gs, M_DBlender_Size);
M_DBlender(genv->gs, 0, 0, genv->d->width,
           genv->d->height);

/* Draw the model */
firstPod = M_Sort(modelPodCount, gBSDF->pods, NULL);
M_Draw(firstPod, app->close);

/* Now draw the background. We could have just appended them
   together above, but this is how to do it when you need to
   render multiple pod lists. */

M_Draw(&gCornerPod, app->close);

/* Here's where the cmd list is sent to the Triangle Engine */
genv->gs->gs_SendList(genv->gs);

/* Once it's rendered, show the bitmap */
GS_WaitIO(genv->gs);
ModifyGraphicsItemVA(genv->d->view, VIEWTAG_BITMAP,
                    GS_GetDestBuffer(genv->gs), TAG_END);

printf("Hit anything on the Control Pad to exit...\n");

GetControlPad(1, TRUE, &cped);
}
```

Cleaning Up

Write code that shuts your game down and performs all needed cleanup functions when the user terminates the program.

This is easy; call

```
M_End(pclosedata);
```

and you're done.

Summary

This chapter showed you exactly how to write a simple Mercury program. Appendix A, "Parts of the Mercury Engine," describes the syntax of many Mercury function calls and provides more details about how they are used in Mercury programs. For still more information about the function calls, macros, and data structures used in Mercury, see this book's appendices and your on-line Mercury documentation.

Parts of the Mercury Engine

This appendix lists and describes the function calls provided in the Mercury rendering engine, arranged by functionality. For more detailed descriptions of the functions listed in this appendix, see your online Mercury documentation.

This appendix is divided into the following major sections:

Topic	Page
The Core Functions	47
Utility Functions	49
Matrix_Perspective	50
M_Light. . . Functions	51
Texture-Blender and Destination-Blender Setup Macros	57
Case Setup Functions	58

The Core Functions

This section lists the four core function calls provided in the Mercury API: `M_Init`, `M_End`, `M_Sort`, and `M_Draw`. These functions were introduced in the section headed “Mercury’s Four Core Functions” on page 4.

M_Init

`M_Init` initializes the system for rendering. It must be called once before any other Mercury functions are called. It allocates, initializes and returns a `CloseData` structure that can subsequently be used for rendering.

The syntax of the `M_Init` function is:

```
CloseData * M_Init (uint32 nverts, uint32 clistsize, GState *gs)
```

Arguments expected by `M_Init` are:

<code>nverts</code>	the maximum number of vertices to be used in a POD.
<code>clistsize</code>	the maximum number of words to be used in a command list.
<code>gs</code>	the <code>GState</code> object used to send command lists to the TE.

M_End

The `M_End` function frees all memory previously allocated by the `M_Init` function, including the memory allocated to the `CloseData` structure initialized by `M_Init`. This is the syntax of the `M_End` function:

```
void M_End(CloseData *pclosedata)
```

`M_End` takes one parameter: `pclosedata`, a value returned by a previous call to `M_Init`.

M_Sort

The `M_Sort` function sorts the linked list of PODs passed in according to each field in the POD structure. The linked list of PODs that exists prior to a call to `M_Sort` may be arranged in a different order when `M_Sort` returns. `M_Sort` returns a pointer to the first pod in the sorted list

The syntax of the `M_Sort` function is:

```
Pod * M_Sort (uint32 count, Pod *podlist, Pod **plastpod)
```

Arguments expected by `M_Sort` are:

<code>count</code>	the number of PODs passed in.
<code>podlist</code>	the first POD in a linked list of PODs.
<code>plastpod</code>	pointer to a pointer to the last POD <RETURNED>.

M_Draw

The `M_Draw` function takes a linked list of PODs as inputs, and outputs the corresponding M2 command list for rendering the PODs. It is the responsibility of your application to send all other M2 command lists for setting up the state of the M2 Triangle Engine state and for producing any effects. The command list produced by `M_Draw` consists of copies of command lists that setup the triangle engine for a case, load or select textures, supplied in the POD data structures, and vertex commands for drawing triangles.

`M_Draw` returns the number of triangles that actually are drawn to the screen. This number does not include rejected triangles. But it does include triangles created as a result of clipping.

The syntax of the `M_Draw` function is:

```
uint32 M_Draw (Pod *firstpod, CloseData *pclose)
```

Parameters expected by the `M_Draw` function are:

<code>firstpod</code>	The first pod in a linked list of PODs in which all the changed flags are set as per <code>M_Sort</code> .
-----------------------	--

pclose

A CloseData area which has been properly initialized.

Utility Functions

Along with the four core functions described under the previous heading, Mercury provides three utility functions that perform lighting, bounds-testing, and texture-related operations.

This section describes each of Mercury's utility functions under a separate heading.

M_Prelight

For each POD in a linked list provided to Mercury, the `M_Prelight` function computes the *rgb* values that result from the lighting calculation of the affected POD and places the result in a corresponding POD in your application's destination list. The source POD provided to the `M_Prelight` function must have normal vectors in its geometry data fields. When the `M_Prelight` function returns, the destination POD has *rgb* values in its geometry data fields.

The source and destination POD list can be the same. However, if you prelight the same geometry twice, the normals that are now colors will yield unexpected results.

This is the syntax of the `M_Prelight` function:

```
void M_Prelight (Pod* src, Pod* dst, CloseData* pclose)
```

Arguments expected by `M_Prelight` are:

src	source POD list
dst	destination POD list
pclose	CloseData structure to use.

M_BoundsTest

The `M_BoundsTest` function tests each bounding box in a list of boxes and sets the flags field to indicate whether the box is visible or not. This is the syntax of the `M_BoundsTest` function:

```
void M_BoundsTest (BBoxList* bbox, CloseData* pclose)
```

`M_BoundsTest` expects the following parameters:

bbox	A list of bounding boxes to test
pclose	CloseData to use

M_LoadPodTexture

By default, the texture reader places the `M_LoadPodTexture` in the `proutine` field of a `PodTexture` structure to load a texture. If you want to override this function, you can replace this function with your own.

```
typedef struct PodTexture
{
    void (*proutine)(void);
    struct TpageSnippets *ptpagesnippets;
    uint32 *ptexture;
    uint32 texturecount;
```

```
uint32 texturebytes;
uint32 *ppip;
uint32 pipbytes;
} PodTexture;
```

M_SetCamera

M_SetCamera takes a normal camera matrix in world space and multiplies it with a specially formatted skew matrix to form Mercury's internal camera skew matrix.

The syntax of the M_SetCamera function is:

```
void M_SetCamera( CloseData *close, Matrix *normalCameraMatrix,
                 Matrix *specialSkewMatrix )
```

Arguments expected by M_SetCamera are:

close	a pointer to Mercury's close data structure
NormalCameraMatrix	a pointer to a matrix that describes the camera's position in world space
SpecialSkewMatrix	a pointer to a specially formatted matrix that contains perspective and screen projection information. You can create this matrix by calling Matrix_Perspective (see next entry).

Matrix_Perspective

The Matrix_Perspective function creates a specially formatted skew matrix using a basic description of the frame buffer geometry and viewing frustum.

The syntax of the Matrix_Perspective function is,

```
void Matrix_Perspective( Matrix *specialSkewMatrix, ViewPyramid
                        *view, float screenXMin, float screenXMax, float screenYMin,
                        float screenYMax, float scaleW )
```

Arguments expected by Matrix_Perspective are:

specialSkewMatrix	a pointer to a Matrix that is used to return the calculated matrix
View	a pointer to a view pyramid. (The view pyramid describes the basic view frustum; (see the <i>vp.</i> calls in Example 2-3 on Page 41.)
ScreenXMin	the minimum screen X position
ScreenXMax	the maximum screen X position
ScreenYMin	the minimum screen Y position
ScreenYMax	the maximum screen Y position
ScaleW	a scaler for <i>w</i> .

M_Draw. . . Functions

The name of each draw function used in Mercury begins with the letters M_Draw. For example, the M_DrawDynLit function dynamically lights triangles according to the list of lights in each POD. (In this manual, this set of functions is referred to generically as Mercury's M_Draw. . . functions. This group of function's should

be distinguished from Mercury's main `M_Draw` function, which is not followed by an ellipsis (. . .) when it appears in this volume.)

This section lists and describes Mercury's `Draw . . .` functions.

M_DrawDynLit

The `M_DrawDynLit` function dynamically lights triangles according to the list of lights in each POD. It outputs triangle commands with vertices containing x , y , r , g , b , and w .

The output alpha of the `M_DrawDynLit` function can represent transparency or fog.

M_DrawDynLitTex

`M_DrawDynLitTex` dynamically lights triangles according to the list of lights in each POD. It outputs triangle commands with vertices containing x , y , r , g , b , a , u , v , and w .

The output alpha can represent transparency or fog.

M_DrawDynLitTrans

The `M_DrawDynLitTrans` function dynamically lights triangles according to the list of lights in each POD. It outputs triangle commands with vertices containing x , y , r , g , b , a , u , v , and w .

The output alpha represents transparency. The output u represents fog.

M_DrawPreLit

The `M_DrawPreLit` function does not light triangles. It obtains the lighting color that is precomputed and already present in the input data. It outputs triangle commands with vertices containing x , y , r , g , b , and w . The output alpha can represent transparency or fog.

M_DrawPreLitTex

`M_DrawPreLitTex` is another function that does not light triangles. It obtains the lighting color already pre-computed from the input data. It outputs triangle commands with vertices containing x , y , r , g , b , a , u , v , and w . The output alpha can represent transparency or fog.

M_DrawPreLitTran

`M_DrawPreLitTran` is still another function that does not light triangles. It obtains the lighting color already pre-computed from the input data. It outputs triangle commands with vertices containing x , y , r , g , b , a , u , v , and w . The output alpha represents transparency. The output u represents fog.

M_Light. . . Functions

The light functions in Mercury are multi-purpose routines. Besides using Mercury's light functions for lighting, you can use them to perform specific per-vertex functions such as creating fog. For more detailed information about how to use Mercury's multi-purpose lighting functions, see the section headed "Mercury Lighting" on page 24.

All the light functions used in Mercury have names that begin with the letters `M_Light`. For example, the `M_LightFog` function computes an alpha value that can be used for a fogging effect. (In this manual, these functions are referred to as Mercury's `M_Light...` functions.)

Initializing Mercury's Light Functions

As explained in "Mercury Lighting" on page 24, Mercury's light functions work a little differently from the other kinds of functions implemented in the Mercury API. First you construct a light list (see "Light Lists" on page 25), and then you supply Mercury with two other kinds of data structures for each POD you want to light: a lighting structure that describes the kind of lighting you want to create for the POD, and a Material structure that describes the kind of material that is being illuminated. When you have provided these two structures, Mercury has all it needs to call the appropriate lighting functions.

To provide Mercury with a light list, you use a `LightList` structure (described in "Light Lists" on page 25). Each entry in a `LightList` structure is made up of two elements: a pointer to a function and a pointer to a second data structure that describes the kind of lighting being used by the corresponding function. This information is provided on a per-vertex basis. Each function that is referenced in a `LightList` branches to a subroutine that initializes the specified lights on a per-object basis. But you do not have to be concerned with these internal details when you construct a light list. All you have to do is provide Mercury with the name of the lighting function you want to use and the name of the light structure that is associated with the function you have selected. Mercury then accesses the lighting function and its associated light structure and performs all necessary lighting operations automatically.

Mercury's Light Functions Listed and Described

Under the headings that follow, Mercury's lighting functions are listed and described.

M_LightFog

The `M_LightFog` function computes an alpha value that is used for a fogging effect. The fog is a linear function on w , where 1.0 shows the object fully and 0.0 is completely fogged. Anything that is at or beyond `fogfar` will be completely fogged. Anything closer than `fognear` is unfogged. To specify the fogging range, Mercury uses a pair of values called `dist1` and `dist2`, which the user can calculate using the formula presented later in this section under the heading

Input: The input to the `M_LightFog` function is a `LightFog` structure:

```
typedef struct LightFog
{
    float dist1, dist2;
}LightFog;
```

Fields: The fields of the `LightFog` structure are:

`dist1, dist2` fog distances

Per-vertex processing: In per-vertex processing, if the fog is 1 at `fognear` and is 0 at `fogfar`, the alpha value is computed as follows:


```

dist1 = hither/(fogfar - fognear)
dist2 = fognear/(fogfar - fognear)

prelimvalue = dist1 * w - dist2,
clamp(prelim value)
alpha -= prelimvalue

```

M_LightDir

The `M_LightDir` function approximates illumination by a light source located at infinity (such as the sun) for which all points are equidistant and only the direction predominates in the calculation. The light intensity is a negated dot product between the light direction and the vertex normal.

Input: The input to the `M_LightDir` function is a `LightDir` structure:

```

typedef struct LightDir
{
    float nx, ny, nz;
    Color3 lightcolor;
} LightDir;

```

Fields: The fields of the `LightDir` structure are:

<code>nx, ny, nz</code>	light direction
<code>lightcolor</code>	light color

Per-vertex processing: In per-vertex processing, the `M_LightDir` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```

color += clamp(-(Nl•Nv))*lightcolor

```

M_LightPoint

`M_LightPoint` simulates a point light source whose light decreases with distance. The light comes from a point, and declines based on inverse square of the distance. The light intensity is a negated dot product between the light direction and the vertex normal.

The `maxdist` value is determined by multiplying intensity times color depth per component times the brightest components value. For 32-bit color each component is 8 bits long. Consequently, you multiply by 256. If you use 16-bit mode with a light of color 0.5, 0.2, 0.2 and an intensity of 25000, `maxdist` equals $25,000 * 32 * 0.5$.

Input: The input to the `M_LightPoint` function is a `LightPoint` structure:

```

typedef struct LightPoint
{
    float x, y, z;
    float maxdist;
    float intensity;
    Color3 lightcolor;
} LightPoint;

```

Fields: The fields of the `LightPoint` structure are:

<code>x, y, z</code>	light position
<code>maxdist</code>	maximum distance light is visible from
<code>intensity</code>	light intensity
<code>lightcolor</code>	light color

Per-vertex processing: In per-vertex processing, the `M_LightPoint` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
Nl = (Pl - Pv) / len(Pl - Pv)
color += clamp(Nl•Nv)*(intensity/distance2)*lightcolor
```

M_LightSoftSpot

When you create lighting using the `M_LightSoftSpot` function, the light you create originates from a point, and radiates in the direction of a normalized vector. The light intensity declines linearly, based on the dot product between the light direction and the light source to object point normalized vector. The light intensity is a negated dot product between the light direction and the vertex normal.

The `maxdist` value is determined by multiplying intensity by color depth per component by the brightest component's value. For 32-bit color, each component is 8 bits long; consequently, you multiply by 256. If you use 16-bit mode with a light of color 0.5, 0.2, 0.2 and an intensity of 25,000, `maxdist` equals $25000 * 32 * 0.5$.

Input: The input to the `M_LightSoftSpot` function is a `M_LightSoftSpot` structure:

```
typedef struct LightSoftSpot
{
    float x, y, z;
    float nx, ny, nz;
    float maxdist;
    float intensity;
    float cos, invcos;
    Color3 lightcolor;
} LightSoftSpot;
```

Fields: The fields of the `M_LightSoftSpot` structure are:

<code>x, y, z</code>	light position
<code>nx, ny, nz</code>	light direction
<code>maxdist</code>	maximum distance light is visible from
<code>intensity</code>	light intensity
<code>cos, invcos</code>	determine the start and end of fadeout region
<code>lightcolor</code>	light color

Per-vertex processing: In per-vertex processing, the `M_LightSoftSpot` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
angle1 = cos(outer angle)
angle2 = 1.0/(cos(inner angle)- cos(outer angle))

Nl = (Pl - Pv) / len(Pl - Pv)
color += clamp((Nl*Nv)-cos)*invcos
        *(intensity/distance2)*lightcolor
```

M_LightFogTrans

The `M_LightFogTrans` function computes a *u* coordinate value that is used for a fogging effect with a transparent nontextured object. The fog is a linear function on *w*, where 1.0 shows the object fully and 0.0 is completely fogged. The lighting calculation computes a *ucoord* value between 0 and 1 that is later scaled to be between 0 and 16.

Input: The input to the `M_LightFogTrans` function is a `LightFog` structure:

```
typedef struct LightFog
{
    float dist1, dist2;
} LightFog;
```

`dist1, dist2` fog distances

Fields: The fields of the `LightFog` structure are:

`dist1, dist2` fog distances

Per-vertex processing: In per-vertex processing, the `M_LightFogTrans` function computes a *u* coordinate which is used for a fogging effect.

If the fog is 1 at `fognear` and 0 is at `fogfar`, the alpha value is computed as follows:

```
dist1 = hither/(fogfar - fognear)
dist2 = fognear/(fogfar - fognear)

prelim value = dist1 * w - dist2,
    which is 0 at fognear and 1 at fogfar
clamp prelim value to 0.0
ucoord -= prelim value
```

M_LightDirSpec

The `M_LightDirSpec` function approximates illumination by a light source located at infinity (such as the sun) for which all points are equidistant and only the direction predominates in the calculation. The light intensity is a negated dot product between the light direction and the vertex normal.

`M_LightDirSpec` also adds a specular reflection to each vertex. It does this by adding a calculated specular color to the pixel's color. The object's specular

properties are described in the object's material properties. The specularFLAG must be set in the POD data structure to inform the initialization code that it needs to transform the camera into the object's local coordinates.

Input: The input to the M_LightDirSpec function is a LightDire structure:

```
typedef struct LightDir
{
    float nx, ny, nz;
    Color3 lightcolor;
} LightDir;
```

Fields: The fields of the LightDir structure are:

nx, ny, nz	light direction
lightcolor	light color

Per-vertex processing: In per-vertex processing, the M_LightDirSpec function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
color += clamp(-(Nl•Nv))*lightcolor
Nc = (Pv - Pc)/ len(Pv - Pc)
Nr = ((2*(Nl•Nv))*Nv)-Nl
Ns = Nc•Nr
specular = Ns/(shine-shine*Ns+Nc)
color += specular * specularcolor
```

M_LightDirSpecTex

The M_LightDirSpecTex function approximates illumination by a light source located at infinity (such as the sun) for which all points are equidistant and only the direction predominates in the calculation. The light intensity is a negated dot product between the light direction and the vertex normal.

M_LightDirSpecTex also adds a specular reflection to each vertex. It does this by adding a constant specular color, scaled by alpha, to the pixel's color in the destination blender. The object's specular properties are described in the object's material properties. The specularFLAG must be set in the POD data structure to inform the initialization code that it needs to transform the camera into the object's local coordinates.

Input: The input to the M_LightDirSpecTex function is a LightDir structure:

```
typedef struct LightDir
{
    float nx, ny, nz;
    Color3 lightcolor;
} LightDir;
```

Fields: The fields of the LightDir structure are:

nx, ny, nz	light direction
lightcolor	light color

Per-vertex processing: In per-vertex processing, the `M_LightDirSpecTex` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
color += clamp(-(Nl•Nv))*lightcolor
Nc = (Pv - Pc) / len(Pv - Pc)
Nr = ((2*(Nl•Nv))*Nv)-Nl
Ns = Nc•Nr
specular = Ns/(shine-shine*Ns+Nc)
alpha += specular
```

Texture-Blender and Destination-Blender Setup Macros

In Mercury, the command lists used by a case are assembled using a set of macros that are designed to set up the Texture Blender and the Destination Blender. These macros are collectively referred to as command-list setup macros.

When you call Mercury's texture-blender setup macros and destination-blender setup macros, the inputs to the Texture Blender are the Gouraud-interpolated colors and alpha calculated using a primitive, a supplied texture color, a supplied alpha, and a constant. The inputs to the Destination Blender are the outputs from the Texture Blender, a source frame buffer and a constant.

This section lists and describes each of Mercury's command-list setup macros.

M_TBNoTex

When you call the `M_TBNoTex` macro, the Texture Blender outputs primitive color and alpha.

M_TBLitTex

When your application calls `M_TBLitTex`, the Texture Blender multiplies the primitive color by the looked up texture color. The primitive alpha is passed through unchanged.

M_TBTex

When `M_TBTex` is called, the Texture Blender passes only looked up texture color and alpha.

M_TBFog

When Mercury executes the `M_TBFog` macro, the Texture Blender linearly interpolates between primitive color and a constant holding the color of the fog. The `u` texture coordinate determines the amount of the blend.

M_DBNoBlend

The `M_DBNoBlend` macro causes Mercury to disable all blending the Destination Blender. The resulting output is the output of the Texture Blender.

M_DBFog

When `M_DBFog` is called, then Destination Blender linearly interpolates between the output from the Texture Blender and a constant holding the color of the fog. The alpha passed in from the Texture Blender determines the amount of the blend.

M_DBSpec

When the `M_DBSpec` macro is executed, the Destination Blender adds the output from the Texture Blender to the product of the specular highlight color times the alpha value output from the Texture blender.

M_DBTrans

When `M_DBTrans` is called, Destination Blender linearly interpolates between the output of the Texture Blender and a corresponding pixel in a source frame buffer. The alpha value passed in from the Texture Blender determines the amount of the blend.

M_DBInit

The `M_DBInit` macro initializes the destination blender state for use with other `M_DB` calls. This must be called once per frame to set the hardware in a known state.

Case Setup Functions

In Mercury, the *case* of a POD is the combination of the state in the triangle engine and the draw function used to construct the triangle commands. Each case used in Mercury is equipped with a small setup function that is called when the first POD of the case is encountered—that is, when the *samecaseFLAG* is not set in the POD. When a setup function is called, it simply places in the `CloseData` structure the draw function that all PODs used by, and then loads a command list into the Triangle Engine.

Mercury provides case functions for 15 predefined cases that can be obtained from combinations of the following prefixes:

<code>dynlit</code> <code>prelit</code>	Two types of lighting (dynamic lighting or prelighting)
<code>fog</code> <code>spec</code> <code>none</code>	Three types of constant color blending (fog, specular, or none)
<code>trans</code> <code>none</code>	either transparent or opaque
<code>tex</code> <code>none</code>	either textured or non-textured.

Some possible combinations—for example `fog + trans + tex`—are not valid because there is no corresponding configuration of the blending stages in the M2 hardware.

The setup functions used in Mercury are listed in Table A-1.

Table A-1 *Combinations of Predefined Cases*

Setup Function	Draw Function	Texture Blender Command	Destination Blender Command
<code>M_SetupDynLit</code>	<code>M_DrawDynLit</code>	<code>M_TBNoTex</code>	<code>M_DBNoBlend</code>
<code>M_SetupDynLitTex</code>	<code>M_DrawDynLitTex</code>	<code>M_TBLitTex</code>	<code>M_DBNoBlend</code>
<code>M_SetupDynLitTrans</code>	<code>M_DrawDynLit</code>	<code>M_TBNoTex</code>	<code>M_DBTrans</code>
<code>M_SetupDynLitTransTex</code>	<code>M_DrawDynLitTex</code>	<code>M_TBLitTex</code>	<code>M_DBTrans</code>

M_SetupDynLitFog	M_DrawDynLit	M_TBNoTex	M_DBFog
M_SetupDynLitFogTex	M_DrawDynLitTex	M_TBLitTex	M_DBFog
M_SetupDynLitFogTrans	M_DrawDynLitTrans	M_TBFog	M_DBTrans
M_SetupDynLitSpecTex	M_DrawDynLitTex	M_TBLitTex	M_DBSpec
M_SetupPreLit	M_DrawPreLit	M_TBNoTex	M_DBNoBlend
M_SetupPreLitTex	M_DrawPreLitTex	M_TBLitTex	M_DBNoBlend
M_SetupPreLitTrans	M_DrawPreLit	M_TBNoTex	M_DBTrans
M_SetupPreLitTransTex	M_DrawPreLitTex	M_TBLitTex	M_DBTrans
M_SetupPreLitFog	M_DrawPreLit	M_TBNoTex	M_DBFog
M_SetupPreLitFogTex	M_DrawPreLitTex	M_TBLitTex	M_DBFog
M_SetupPreLitFogTrans	M_DrawPreLitTrans	M_TBFog	M_DBTrans

Matrix Calls

This appendix provides a complete list of reference pages for each Matrix call. The calls included in this appendix, and a brief description, are listed below.

Matrix_	Computes a perspective matrix that is concatenated on to the camera matrix.
Matrix_BillboardX	Calculates a matrix that lays perpendicular to a vector about it's X-axis
Matrix_BillboardY	Calculates a matrix that lays perpendicular to a vector about it's Y-axis
Matrix_BillboardZ	Calculates a matrix that lays perpendicular to a vector about it's Z-axis
Matrix_Construct	Allocates an instance of a matrix and sets it to identity.
Matrix_Copy	Copyies from a source matrix to a detination matrix.
Matrix_CopyOrientation	Copy the orientation of a source matrix to a destination matrix.
Matrix_CopyTranslation	Copy the translation of a source matrix to a destination matrix.
Matrix_Destruct	Frees the givem matrix from memory.
Matrix_FullInvert	Full matrix invert.
Matrix_GetBank	Calculates the heading component of a matrix.
Matrix_GetElevation	Calculates the elevation component of a matrix.
Matrix_GetHeading	Calculates the heading component of a matrix.
Matrix_GetTranslation	Copy the transformation information from the matrix to a vector.
Matrix_GetpTranslation	Get a pointer to the translation within a matrix.

Matrix_GetTranslation	Sets the matrix to identity.
Matrix_Invert	Matrix invert.
Matrix_LookAt	Calculates a matrix that looks down the -z-axis at a specified point.
Matrix_LookAt	Translate matrix by three floats in local orientation.
Matrix_MoveByVector	Translate matrix by vector in local orientation.
Matrix_Mult	Matrix multiply two source matrices.
Matrix_Multiply	Multiplies a source matrix with the destination matrix.
Matrix_MultiplyOrientation	Multiplies the orientation section of a source matrix with the destination matrix.
Matrix_MultOrientation	Matrix multiply the orientation section of two source matrices.
Matrix_Normalize	Normalize the given matrix.
Matrix_Print	Print the matrix.
Matrix_Rotate	Rotate the matrix in world space.
Matrix_RotateLocal	Rotate the orientation of a matrix in world space.
Matrix_RotateX	Rotate the orientation of a matrix in world space about the X-axis.
Matrix_RotateXLocal	Rotate the matrix in world orientation and in local space about the X-axis.
Matrix_RotateY	Rotate the matrix in world space about the X-axis.
Matrix_RotateYLocal	Rotate the matrix in world orientation and in local space about the Y-axis.
Matrix_RotateZ	Rotate the orientation of a matrix in world space about the Z-axis.
Matrix_RotateZLocal	Rotate the matrix in world orientation and in local space about the Z-axis.
Matrix_Scale	Scale matrix by three floats.
Matrix_ScaleByVector	Scales the matrix by a vector.
Matrix_ScaleLocal	Scale matrix in local coordinates by three floats.
Matrix_ScaleLocalByVector	Scale matrix in local coordinates by a vector.
Matrix_SetTranslation	Set the matrix's translation.
Matrix_SetTranslationByVector	Set a matrix's translation from a vector.
Matrix_Translate	Translate the matrix by three floats.
Matrix_TranslateByVector	Translate the matrix by a vector.
Matrix_Turn	Rotate the matrix in local space.

Matrix_TurnLocal	Rotate the matrix in local space updating the orientation only.
Matrix_TurnX	Rotate the matrix in local space about the X-axis.
Matrix_TurnXLocal	Rotate the matrix in local space about the X-axis updating the orientation only.
Matrix_TurnY	Rotate the matrix in local space about the Y-axis.
Matrix_TurnYLocal	Rotate the matrix in local space about the Y-axis updating the orientation only.
Matrix_TurnZ	Rotate the matrix in local space about the Z-axis.
Matrix_TurnZLocal	Rotate the matrix in local space about the Z-axis updating the orientation only.
Matrix_Zero	Zeros the contents of a matrix.

Matrix_

Computes a perspective matrix that is concatenated on to the camera matrix.

Synopsis

```
void Matrix_Perspective(pMatrix matrix, pViewPyramid p, float
screen_xmin, float screen_xmax, float screen_ymin, float
screen_ymax, float wscale)
```

Description

Computes a perspective matrix that is concatenated on to the camera matrix.

Arguments

matrix	Pointer to the destination Matrix structure.
p	Pointer to the: screen_xmin value. screen_xmax value. screen_ymin value. screen_ymax value. wscale value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_BillboardX

Calculates a matrix that lays perpendicular to a vector about its X-axis

Synopsis

```
void Matrix_BillboardX( Matrix *m, Vector3D *v );
```

Description

Calculates a matrix that lays perpendicular to a vector about its X-axis

Arguments

m	Pointer to the destination Matrix structure.
v	Pointer to a vector to 'lookat'.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_BillboardY

Calculates a matrix that lays perpendicular to a vector about its Y-axis

Synopsis

```
void Matrix_BillboardY( Matrix *m, Vector3D *v );
```

Description

Calculates a matrix that lays perpendicular to a vector about its Y-axis

Arguments

m	Pointer to the destination Matrix structure.
v	Pointer to a vector to 'lookat'.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(), Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(), Matrix_MultiplyOrientation(), Matrix_Zero(), Matrix_Identity(), Matrix_TranslateByVector(), Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(), Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(), Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(), Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_BillboardZ

Calculates a matrix that lays perpendicular to a vector about its Z-axis

Synopsis

```
void Matrix_BillboardZ( Matrix *m, Vector3D *v );
```

Description

Calculates a matrix that lays perpendicular to a vector about its Z-axis

Arguments

m	Pointer to the destination Matrix structure.
v	Pointer to a vector to 'lookat'.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(), Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(), Matrix_MultiplyOrientation(), Matrix_Zero(), Matrix_Identity(), Matrix_TranslateByVector(), Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(), Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(), Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(), Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_Construct

Allocates an instance of a matrix and sets it to identity.

Synopsis

```
Matrix* Matrix_Construct( void );
```

Description

Allocates an instance of a matrix and sets it to identity.

Arguments

void

Return Value

Matrix* Pointer to the new Matrix structure.

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_Copy

Copies from a source matrix to a detination matrix.

Synopsis

```
void Matrix_Copy( Matrix *m, Matrix *a ); Matrix*  
Matrix_Construct( void );
```

Description

Copies from a source matrix to a detination matrix.

Arguments

m*	Pointer to the destination Matrix structure.
a*	Pointer to the source Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_CopyOrientation

Copy the orientation of a source matrix to a destination matrix.

Synopsis

```
void Matrix_CopyOrientation( Matrix *m, Matrix *b );
```

Description

Copy the orientation of a source matrix to a destination matrix.

Arguments

m	Pointer to the destination Matrix structure.
b	Pointer to a source matrix.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_CopyTranslation

Copy the translation of a source matrix to a destination matrix.

Synopsis

```
void Matrix_CopyTranslation( Matrix *m, Matrix *b );
```

Description

Copy the translation of a source matrix to a destination matrix.

Arguments

m	Pointer to the destination Matrix structure.
b	Pointer to a source matrix.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_Destruct

Frees the given matrix from memory.

Synopsis

```
void Matrix_Destruct( Matrix *m );
```

Description

Frees the given matrix from memory.

Arguments

m* Pointer to the Matrix structure to be deleted.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_FullInvertFull matrix invert.

Synopsis

```
Err Matrix_FullInvert(pMatrix dst, pMatrix src);
```

Description

The code is based on the code presented in "Graphics Gems" (ed. Andrew Glassner), in "Matrix Inversion" (pg. 766). The code has been modified to work with the 3x4 matrices we use. We don't have a fourth column, so it is hardwired as (0, 0, 0, 1). Print the matrix.

Arguments

dst	Pointer to the destination Matrix structure.
src	Pointer to the source Matrix structure.

Return Value**Err****Implementation**

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_GetBank

Calculates the heading component of a matrix.

Synopsis

```
float Matrix_GetBank( Matrix *m );
```

Description

Calculates the banking component of a matrix.

Arguments

m Pointer to the destination Matrix structure.

Return Value

float angle in radians. 0 is aligned along the +x axis. Positive angles are to the righthand side up. Negative angles are to the righthand side down.

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Synopsis

Description

Arguments

Return Value

Implementation

Associated Files

See Also

MER ♦ 75

Matrix_GetHeading

Calculates the heading component of a matrix.

Synopsis

```
float Matrix_GetHeading( Matrix *m );
```

Description

Calculates the heading component of a matrix.

Arguments

m Pointer to the destination Matrix structure.

Return Value

Float angle in radians. 0 is aligned along the -z axis. Positive angles are to the righthand side. Negative angles are to the lefthand side.

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_GetTranslation

Copy the transformation information from the matrix to a vector.

Synopsis

```
void Matrix_GetTranslation( Matrix *m, Vector3D *v );
```

Description

Copy the transformation information from the matrix to a vector.

Arguments

m*	Pointer to the destination Matrix structure.
v*	Pointer to the destination Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Construct(), Matrix_GetpTranslation(),
Matrix_SetTranslationByVector(), Matrix_SetTranslation(),
Matrix_TranslateByVector(), Matrix_Translate(), Matrix_Move()

Matrix_GetpTranslation

Get a pointer to the translation within a matrix.

Synopsis

```
Vector3D* Matrix_GetpTranslation( Matrix *m );
```

Description

Get a pointer to the translation within a matrix.

Arguments

<code>m*</code>	Pointer to the destination Matrix structure.
-----------------	--

Return Value

<code>v*</code>	Pointer to the matrix's translation value.
-----------------	--

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

`Matrix_Construct()`, `Matrix_GetTranslation()`,
`Matrix_SetTranslationByVector()`, `Matrix_SetTranslation()`,
`Matrix_Translate()`, `Matrix_MoveByVector()`, `Matrix_Move()`

Matrix_IdentitySets the matrix to identity.

Synopsis

```
Matrix_Identity( nMatrix *m );
```

Description

Sets the matrix to identity.

Arguments

m* Pointer to the destination Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_Invert

Matrix invert.

Synopsis

```
void Matrix_Invert( Matrix *dst, Matrix *src );
```

Description

Matrix invert.

Arguments

dst	Pointer to the destination Matrix structure.
src	Pointer to the source Matrix structure.

Return Value

Err

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_LookAt

Calculates a matrix that looks down the -z-axis at a specified point.

Synopsis

```
void Matrix_LookAt(Matrix *m, Vector3D *pWhere, Vector3D
*pMe, float twist);
```

Description

Calculates a matrix that lays perpendicular to a vector about its Y-axis

Arguments

m	Pointer to the destination Matrix structure.
pWhere	Pointer to a value to look at.
pMe	Pointer to a value to look from.
twist	A rotation in radians away from the up vector.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

Matrix_Move

Translate matrix by three floats in local orientation.

Synopsis

```
void nMatrix_Move( nMatrix *m, float x, float y, float z );
```

Description

Translate matrix by three floats in local orientation.

Arguments

m*	Pointer to the destination Matrix structure.
x	Move X value.
y	Move Y value.
z	Move Z value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_MoveByVectorTranslate matrix by vector in local orientation.

Synopsis

```
void nMatrix_MoveByVector( nMatrix *m, nVector3D *v );
```

Description

Translate matrix by vector in local orientation.

Arguments

m*	Pointer to the destination Matrix structure.
v*	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_Mult

Matrix multiply two source matrices.

Synopsis

```
void nMatrix_Mult( nMatrix *m, nMatrix *a, nMatrix *b );
```

Description

Matrix multiply two source matrices. [m =a x b]

Arguments

m*	Pointer to the destination Matrix structure.
a*	Pointer to the first source Matrix structure.
a*	Pointer to the second source Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_MultiplyMultiplies a source matrix with the destination matrix.

Synopsis

```
void Matrix_Multiply( Matrix *m, Matrix *a );
```

Description

Multiplies a source matrix with the destination matrix. [m =m x a]

Arguments

m*	Pointer to the destination Matrix structure.
a*	Pointer to the source Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_MultiplyOrientation

Multiplies the orientation section of a source matrix with the destination matrix.

Synopsis

```
void nMatrix_MultiplyOrientation( nMatrix *m, nMatrix *a );
```

Description

Multiplies the orientation section of a source matrix with the destination matrix.
[m =m x a]

Arguments

m*	Pointer to the destination Matrix structure.
a*	Pointer to the source Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_MultOrientation

Matrix multiply the orientation section of two source matrices.

Synopsis

```
void nMatrix_MultOrientation( nMatrix *m, nMatrix *a,
                             nMatrix *b );
```

Description

Matrix multiply the orientation section of two source matrices. [m =a x b]

Arguments

m*	Pointer to the destination Matrix structure.
a*	Pointer to the first source Matrix structure.
b*	Pointer to the second destination Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_Normalize

Normalize the given matrix.

Synopsis

```
void Matrix_Normalize( Matrix *m );
```

Description

Normalize the given matrix.

Arguments

m* Pointer to the destination Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Synopsis

Description

See Also

MER ♦ 89

Matrix_Rotate

Rotate the matrix in world space.

Synopsis

```
void nMatrix_Rotate( nMatrix *m, char axis, float r );
```

Description

Rotate the matrix in world space.

Arguments

m*	Pointer to the destination Matrix structure.
axis	World axis to rotate around ['X','Y' or 'Z'].
r	Rotation angle in radians

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_RotateLocal

Rotate the orientation of a matrix in world space.

Synopsis

```
void nMatrix_RotateLocal( nMatrix *m, char axis, float r );
```

Description

Rotate the orientation of a matrix in world space.

Arguments

m*	Pointer to the destination Matrix structure.
axis	World axis to rotate around ['X','Y' or 'Z'].
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_RotateX

Rotate the orientation of a matrix in world space about the X-axis.

Synopsis

```
void nMatrix_RotateX( nMatrix *m, float r );
```

Description

Rotate the orientation of a matrix in world space about the X-axis.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_RotateXLocal

Rotate the matrix in world orientation and in local space about the X-axis.

Synopsis

```
void Matrix_RotateXLocal( Matrix *m, float r );
```

Description

Rotate the matrix in world orientation and in local space about the X-axis

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(),
Matrix_RotateZ(), Matrix_RotateLocal(),
Matrix_RotateYLocal(), Matrix_RotateZLocal(), Matrix_Turn(),
Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(),
Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal()

Matrix_RotateY

Rotate the orientation of a matrix in world space about the Y-axis.

Synopsis

```
void nMatrix_RotateY( nMatrix *m, float r ); Matrix*  
Matrix_Construct( void );
```

Description

Rotate the orientation of a matrix in world space about the Y-axis.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_RotateYLocal

Rotate the matrix in world orientation and in local space about the Y-axis.

Synopsis

```
void Matrix_RotateYLocal( Matrix *m, float r );
```

Description

Rotate the matrix in world orientation and in local space about the Y-axis

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(),
Matrix_RotateZ(), Matrix_RotateLocal(),
Matrix_RotateXLocal(), Matrix_RotateZLocal(), Matrix_Turn(),
Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(),
Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal()

Matrix_RotateZ

Rotate the orientation of a matrix in world space about the Z-axis.

Synopsis

```
void nMatrix_RotateZ( nMatrix *m, float r );
```

Description

Rotate the orientation of a matrix in world space about the Z-axis.

Arguments

m*	Pointer to the destination Matrix structure.
	Rotation angle in radians

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_RotateZLocal

Rotate the matrix in world orientation and in local space about the Z-axis.

Synopsis

```
void Matrix_RotateZLocal( Matrix *m, float r );
```

Description

Rotate the matrix in world orientation and in local space about the Z-axis

Arguments

<code>m*</code>	Pointer to the destination Matrix structure.
<code>r</code>	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(),
Matrix_RotateZ(), Matrix_RotateLocal(),
Matrix_RotateXLocal(), Matrix_RotateYLocal(), Matrix_Turn(),
Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(),
Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal()

Matrix_Scale

Scale matrix by three floats.

Synopsis

```
void nMatrix_Scale( nMatrix *m, float x, float y, float z );  
Matrix* Matrix_Construct( void );
```

Description

Scale matrix by three floats.

Arguments

m*	Pointer to the destination Matrix structure.
x	Scale X value.
y	Scale y value.
z	Scale Z value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_ScaleByVectorScales the matrix by a vector.

Synopsis

```
void nMatrix_ScaleByVector( nMatrix *m, nVector3D *v );
```

Description

Scales the matrix by a vector.

Arguments

m*	Pointer to the destination Matrix structure.
v*	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_ScaleLocal

Scale matrix in local coordinates by three floats.

Synopsis

```
void Matrix_ScaleLocal( Matrix *m, float x, float y, float z
);
```

Description

Scale matrix in local coordinates by three floats.

Arguments

m*	Pointer to the destination Matrix structure.
x	Scale x value.
y	Scale y value.
z	Scale z value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_ScaleByVector(), Matrix_Scale(),
Matrix_ScaleLocalByVector()

Matrix_ScaleLocalByVectorScale matrix in local coordinates by a vector.

Synopsis

```
void Matrix_ScaleLocalByVector( Matrix *m, Vector3D *v );
```

Description

Scale matrix in local coordinates by a vector.

Arguments

<code>m*</code>	Pointer to the destination Matrix structure.
<code>v*</code>	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_ScaleByVector(), Matrix_Scale(), Matrix_ScaleLocal()

Matrix_SetTranslation

Set the matrix's translation.

Synopsis

```
void Matrix_SetTranslation(Matrix *m, float x, float y, float z);
```

Description

Set the matrix's translation.

Arguments

m*	Pointer to the destination Matrix structure.
x	New x value.
y	New y value.
z	New z value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Construct(), Matrix_GetTranslation(),
Matrix_GetpTranslation(), Matrix_SetTranslationByVector(),
Matrix_TranslateByVector(), Matrix_Translate(),
Matrix_MoveByVector(), Matrix_Move()

Matrix_SetTranslationByVector Set a matrix's translation from a vector.

Synopsis

```
void Matrix_SetTranslationByVector( Matrix *m, Vector3D *v );
```

Description

Set a matrix's translation from a vector.

Arguments

<code>m*</code>	Pointer to the destination Matrix structure.
<code>v*</code>	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

`Matrix_Construct()`, `Matrix_GetTranslation()`,
`Matrix_GetpTranslation()`, `Matrix_SetTranslation()`,
`Matrix_TranslateByVector()`, `Matrix_Translate()`,
`Matrix_MoveByVector()`, `Matrix_Move()`

Matrix_Translate

Translate the matrix by three floats.

Synopsis

```
void nMatrix_Translate( nMatrix *m, float x, float y,
                        float z );
```

Description

Translate the matrix by three floats.

Arguments

m*	Pointer to the destination Matrix structure.
x	Translate X value.
y	Translate Y value.
z	Translate Z value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_TranslateByVectorTranslate the matrix by a vector.

Synopsis

```
void nMatrix_TranslateByVector( nMatrix *m, nVector3D *v );
```

Description

Translate the matrix by a vector.

Arguments

m*	Pointer to the destination Matrix structure.
v*	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_Turn

Rotate the matrix in local space.

Synopsis

```
Matrix_Turn( nMatrix *m, char axis, float r );
```

Description

Rotate the matrix in local space.

Arguments

m*	Pointer to the destination Matrix structure.
axis	World axis to rotate around ['X','Y' or 'Z'].
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_TurnLocal

Rotate the matrix in local space updating the orientation only.

Synopsis

```
void nMatrix_TurnLocal( nMatrix *m, char axis, float r );
```

Description

Rotate the matrix in local space updating the orientation only.

Arguments

<code>m*</code>	Pointer to the destination Matrix structure.
<code>axis</code>	World axis to rotate around ['X','Y' or 'Z'].
<code>r</code>	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_TurnX

Rotate the matrix in local space about the X-axis.

Synopsis

```
void nMatrix_TurnX( nMatrix *m, float r );
```

Description

Rotate the matrix in local space about the X-axis.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_TurnXLocal

Rotate the matrix in local space about the X-axis updating the orientation only.

Synopsis

```
void nMatrix_TurnXLocal( nMatrix *m, float r );
```

Description

Rotate the matrix in local space about the X-axis updating the orientation only.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_TurnY

Rotate the matrix in local space about the Y-axis.

Synopsis

```
void nMatrix_TurnY( nMatrix *m, float r ); Matrix*  
Matrix_Construct( void );
```

Description

Rotate the matrix in local space about the Y-axis.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_TurnYLocal

Rotate the matrix in local space about the Y-axis updating the orientation only.

Synopsis

```
void nMatrix_TurnYLocal( nMatrix *m, float r );
```

Description

Rotate the matrix in local space about the X-axis updating the orientation only.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),  
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),  
Matrix_MultiplyOrientation(), Matrix_Zero(),  
Matrix_Identity(), Matrix_TranslateByVector(),  
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),  
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),  
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),  
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),  
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),  
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

Matrix_TurnZ

Rotate the matrix in local space about the Z-axis.

Synopsis

```
void nMatrix_TurnZ( nMatrix *m, float r ); Matrix*  
Matrix_Construct( void );
```

Description

Rotate the matrix in local space about the Z-axis.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_TurnZLocal

Rotate the matrix in local space about the Z-axis updating the orientation only.

Synopsis

```
void nMatrix_TurnZLocal( nMatrix *m, float r );
```

Description

Rotate the matrix in local space about the Z-axis updating the orientation only.

Arguments

m*	Pointer to the destination Matrix structure.
r	Rotation angle in radians.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(), Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(), Matrix_MultiplyOrientation(), Matrix_Zero(), Matrix_Identity(), Matrix_TranslateByVector(), Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(), Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(), Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(), Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Matrix_Zero

Zeros the contents of a matrix.

Synopsis

```
void nMatrix_Zero( nMatrix *m );
```

Description

Zeros the contents of a matrix.

Arguments

m* Pointer to the destination Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

Vector Calls

This appendix provides a complete list of reference pages for each Matrix call. The calls included in this appendix, and a brief description, are listed below.

Vector3D_Add	Add a source vector to a destination vector.
Vector3D_Average	Calculated the average of two vectors.
Vector3D_Compare	Compare two source vectors.
Vector3D_CompareFuzzy	Compare two source vectors with fuzzy edges.
Vector3D_Construct	Creates an instance of Vector3D.
Vector3D_Copy	Copy a source vector to a destination vector.
Vector3D_Cross	Calculates the cross product between two vectors.
Vector3D_Destruct	Deletes a Vector3D instance.
Vector3D_Dot	Calculates the dot product between two vectors.
Vector3D_GetX	Returns the X component of a Vector3D struct.
Vector3D_GetY	Returns the Y component of a Vector3D struct.
Vector3D_GetZ	Returns the Z component of a Vector3D struct.
Vector3D_Length	Calculated the length of a vector.
Vector3D_Maximum	Find the maximum vector from two source vectors.
Vector3D_Minimum	Find the minimum vector from two source vectors.
Vector3D_Multiply	Multiply two source vectors and place result in a destination vector.
Vector3D_MultiplyByMatrix	Multiplies the vector by a matrix.
Vector3D_Negate	Negates the given vector.
Vector3D_Normalize	Normalize the given vector.

Vector3D_OrientateByMatrix	Multiplies the vector by the orientation of the matrix.
Vector3D_Print	Prints the vector.
Vector3D_Scale	Scale a vector.
Vector3D_Set	Initializes a Vector3D struct.
Vector3D_Subtract	Subtract two vectors and store the result in a destination vector.
Vector3D_Zero	Zeros the vector.

Vector3D_AddAdd Add a source vector to a destination vector.

Synopsis

```
void Vector3D_Add(Vector3D *v, Vector3D *a);
```

Description

Add a source vector to a destination vector.

Arguments

<code>*v</code>	Pointer to the destination Vector3D structure.
<code>*a</code>	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),  
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),  
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),  
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),  
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),  
Vector3D_Zero(), Vector3D_Print()
```

Vector3D_Average

Calculated the average of two vectors.

Synopsis

```
void Vector3D_Average(Vector3D *v, Vector3D *a, Vector3D *b);
```

Description

Calculated the average of two vectors.

Arguments

*v	Pointer to the destination Vector3D structure
*a	Pointer to first source Vector3D structure
*b	

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_CompareCompare two source vectors.

Synopsis

```
bool Vector3D_Compare(Vector3D *v, Vector3D *a);
```

Description

Compare two source vectors.

Arguments

<code>*v</code>	Pointer to the first source Vector3D structure.
<code>*a</code>	Pointer to the second source Vector3D structure.

Return Value

<code>bool</code>	TRUE Both vectors are exactly equal. FALSE Both vectors are not equal.
-------------------	---

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_CompareFuzzyCompare two source vectors with fuzzy edges.

Synopsis

```
bool Vector3D_CompareFuzzy(Vector3D *v, Vector3D *a, float
fuzzy);
```

Description

Compare two source vectors with fuzzy edges.

Arguments

*v	Pointer to the first source Vector3D structure.
*a	Pointer to the second source Vector3D structure.
fuzzy	The fuzzy tolerance level.

Return Value

bool	TRUE Both vectors are exactly equal. FALSE Both vectors are not equal.
------	---

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_ConstructCreates an instance of Vector3D.

Synopsis

```
Vector3D* Vector3D_Construct(void);
```

Description

Creates an instance of Vector3D.

Arguments

void

Return Value

Vector3D* Pointer to the new Vector3D structure.

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_Copy

Copy a source vector to a destination vector.

Synopsis

```
void Vector3D_Copy(Vector3D *v, Vector3D *a);
```

Description

Copy a source vector to a destination vector.

Arguments

*v	Pointer to the destination Vector3D structure.
*a	Pointer to the source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_CrossCalculates the cross product between two vectors.

Synopsis

```
void Vector3D_Multiply(Vector3D *v, Vector3D *a, Vector3D
*b);
```

Description

Calculates the cross product between two vectors and places result in destination vector.

Arguments

<code>*v</code>	Pointer to the destination Vector3D structure.
<code>*a</code>	Pointer to the first source Vector3D structure.
<code>*b</code>	Pointer to the second source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

`Vector3D_Set()`, `Vector3D_Average()`, `Vector3D_Negate()`,
`Vector3D_Dot()`, `Vector3D_Length()`, `Vector3D_Normalize()`,
`Vector3D_Minimum()`, `Vector3D_Maximum()`, `Vector3D_Compare()`,
`Vector3D_Copy()`, `Vector3D_Add()`, `Vector3D_Sub()`,
`Vector3D_Scale()`, `Vector3D_Multiply()`, `Vector3D_Cross()`,
`Vector3D_Zero()`, `Vector3D_Print()`

Vector3D_Destroy

Deletes a Vector3D instance.

Synopsis

```
void Vector3D_Destroy( Vector3D *v );
```

Description

Deletes a Vector3D instance.

Arguments

*v	Pointer to the Vector3D structure to be deleted.
----	--

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_DotCalculates the dot product between two vectors.

Synopsis

```
float Vector3D_Dot(Vector3D *v, Vector3D *a);
```

Description

Calculates the dot product between two vectors.

Arguments

<code>*v</code>	Pointer to the destination Vector3D structure.
<code>*a</code>	Pointer to the source Vector3D structure.

Return Value

float dot product value

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Synopsis

Description

Arguments

Return Value

Implementation

Associated Files

See Also

MER ♦ 126

Synopsis

Description

Arguments

Return Value

Implementation

Associated Files

See Also

MER ♦ 127

Vector3D_GetZ

Returns the Z component of a Vector3D struct.

Synopsis

```
float Vector3D_GetZ(Vector3D *v);
```

Description

Returns the Z component of a Vector3D struct.

Arguments

*v Pointer to a Vector3D structure.

Return Value

float Z value.

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Compare(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Length(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_LengthCalculates the length of a vector.

Synopsis

```
float Vector3D_Length(Vector3D *v);
```

Description

Calculates the length of a vector.

Arguments

*v Pointer to the destination Vector3D structure.

Return Value

float length value.

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_Maximum

Find the maximum vector from two source vectors.

Synopsis

```
void Vector3D_Maximum(Vector3D *v, Vector3D *a, Vector3D *b);
```

Description

Find the minimum vector from two source vectors.

Arguments

*v	Pointer to the destination Vector3D structure.
*a	Pointer to the first source Vector3D structure.
*b	Pointer to the second source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_MinimumFind the minimum vector from two source vectors.

Synopsis

```
void Vector3D_Minimum(Vector3D *v, Vector3D *a, Vector3D *b);
```

Description

Find the minimum vector from two source vectors.

Arguments

<code>*v</code>	Pointer to the destination Vector3D structure.
<code>*a</code>	Pointer to the first source Vector3D structure.
<code>*b</code>	Pointer to the second source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_Multiply

Multiply two source vectors and place result in a destination vector.

Synopsis

```
void Vector3D_Multiply(Vector3D *v, Vector3D *a, Vector3D
*b);
```

Description

Multiply two source vectors and place result in a destination vector.

[v.x = a.x * b.x] [v.y = a.y * b.y] [v.z = a.z * b.z]

Arguments

*v	Pointer to the destination Vector3D structure.
*a	Pointer to the first source Vector3D structure.
*b	Pointer to the second source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(),
Vector3D_MultiplyByMatrix(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_MultiplyByMatrix Multiplies the vector by a matrix..

Synopsis

```
void Vector3D_MultiplyByMatrix( Vector3D *v, Matrix *m );
```

Description

Multiplies the vector by a matrix.

Arguments

<code>*v</code>	Pointer to the destination Vector3D structure.
<code>*m</code>	Pointer to the Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

```
Vector3D_Set(), Vector3D_Average(),  
Vector3D_Negate(), Vector3D_Multiply(), Vector3D_Dot(),  
Vector3D_Length(), Vector3D_Normalize(), Vector3D_Minimum(),  
Vector3D_Maximum(), Vector3D_Compare(), Vector3D_Copy(),  
Vector3D_Add(), Vector3D_Sub(), Vector3D_Scale(),  
Vector3D_Multiply(), Vector3D_Cross(), Vector3D_Zero(),  
Vector3D_Print()
```

Synopsis

Description

Arguments

Return Value

Implementation

Associated Files

See Also

MER ♦ 134

Synopsis

Description

Arguments

Pointer to the destination Vector3D structure.

Return Value

Implementation

Associated Files

See Also

MER ♦ 135

Vector3D_OrientateByMatrix Multiplies the vector by the orientation of the matrix.

Synopsis

```
void Vector3D_OrientateByMatrix( Vector3D *v, Matrix *m );
```

Description

Multiplies the vector by the orientation of the matrix.

Arguments

<code>*v</code>	Pointer to the destination Vector3D structure.
<code>*m</code>	Pointer to the Matrix structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Construct()

Vector3D_Scale

Scale a vector.

Synopsis

```
void Vector3D_Scale( Vector3D *v, float x, float y, float z );
```

Description

Scale a vector in x,y & z. [v.x *= x; v.y *= y; v.z *= z]

Arguments

*v	Pointer to the destination Vector3D structure.
x	X scale value.
y	Y scale value.
z	Z scale value.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_SetInitializes a Vector3D struct.

Synopsis

```
void Vector3D_Set(Vector3D *v, float x, float y, float z);
```

Description

Initializes a Vector3D struct.

Arguments

<code>*v</code>	Pointer to a Vector3D structure.
<code>x</code>	The new X value for the Vector3D.
<code>y</code>	The new Y value for the Vector3D.
<code>z</code>	The new Z value for the Vector3D.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Vector3D_Subtract

Subtract two vectors and store the result in a destination vector.

Synopsis

```
void Vector3D_Subtract( Vector3D *v, Vector3D *a, Vector3D
                        *b );
```

Description

Subtract two vectors and store the result in a destination vector. [v =a -b]

Arguments

*v	Pointer to the destination Vector3D structure.
*a	Pointer to the first source Vector3D structure.
*b	Pointer to the second source Vector3D structure.

Return Value

void

Implementation

Version 1.0

Associated Files

<matrix.h>, mercury/lib

See Also

Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()

Synopsis

Description

Arguments

Return Value

Implementation

Associated Files

See Also

MER ♦ 141

Index

Symbols

. utility function MER-11
.s files MER-11

Numerics

3D geometry MER-13
 defined MER-13
3D graphics pipeline MER-1
3DO Command List Toolkit manual MER-2,
MER-33
3DO Development Environment Installation Guide
MER-vii
3DO M2 Command List Toolkit manual MER-viii
3DO M2 Debugger Programmer's Guide MER-viii
3DO M2 Graphics Programmer's Guide MER-viii
602 CPU MER-1

A

a light li MER-21
Addison-Wesley MER-viii
alpha base value MER-27
ambient light MER-27
ambient material property MER-27
anatomy of a Mercury application MER-36–MER-
46
API MER-1
 Mercury, see Mercury API MER-4
appData->close MER-40

application
 anatomy of MER-36–MER-46
 developing MER-3
 writing MER-37
application programming interface, see API MER-4
assembly language MER-1
atrix 77

B

back face rejection MER-12
backgrounds
 drawing MER-43
 PODs and MER-3
base color MER-27
base field
 in Material structure MER-27
base matrices
 compositing MER-30
bigcircle example program MER-7
blender
 destination, see destination blender MER-3
 texture, see texture blender MER-5
bounding box MER-3, MER-11
bounding boxes
 checking MER-43
bounds-testing MER-5, MER-43
bsdfreader. MER-37

C**C APIs MER-1**

cache alignment MER-1

caching

instruction MER-7

caddr field MER-42

callatendFLAG MER-23

callatstartFLAG MER-22

calls

matrix 61, 61–114

vector 115–141

camera MER-3

creating and setting up MER-41

setting up and initializing MER-36, MER-38

updating MER-13

camera coordinates MER-32

camera matrix MER-43

camera matrix in world space MER-31

camera operations

in helloworld program MER-31

camera placement

in helloworld program MER-37

camera skew matrix MER-30, MER-41

example of MER-30, MER-41

initializing MER-41

camera transformation MER-13, MER-32

case

defined MER-5, MER-11

case code MER-11

case of a POD MER-11

case setup functions MER-5, MER-58

casecodeisasmFLAG MER-22

CFAN flag MER-20

C-language APIs MER-1

cleaning up MER-45

cleanup MER-4

cleanup functions MER-45

clipFLAG MER-23

clipping

vertices MER-12

clockwise triangles MER-17, MER-20

closedata MER-27

CloseData structure MER-11, MER-36

caddr field of MER-42

defined MER-6

described MER-12

fields of MER-12

importance of MER-40

initializing MER-38

setting fields in MER-40

setting up MER-40

CLT functions

see Command List Toolkit functions MER-42

CLT_ClearFrameBuffer MER-44, MER-45

CLT_ClearFrameBuffer function MER-43

color calculations MER-16

colors MER-13

command list

sending to Triangle Engine MER-43

Command List Toolkit (CLT) functions MER-42

command-line arguments

in helloworld program MER-37

command-list buffers MER-38

Computer Graphics Principles and Practice

MER-viii

convlights MER-27

convlights buffer MER-27

coordinate system

M2 MER-13

world MER-30

coordinate transform

in skew matrix formula MER-30

coordinate transformation

in skew matrix formula MER-30

coordinates

camera MER-32

normalizing MER-13

PodGeometry MER-21

world MER-31

core functions MER-4, MER-47

counter-clockwise MER-17, MER-20

counter-clockwise triangles MER-17, MER-20

CPU

602 MER-1

CPU cycles MER-13

creating a game MER-36

cube

constructing MER-14

faceted MER-17

cycles

CPU MER-13

D

d specular lightin MER-28
 data cache MER-10
 data cache alignment MER-1
 data structures MER-6
 data.c MER-3, MER-37
 Davis Open GL Programming Guid MER-viii
 depth field MER-12
 descriptors
 pipeline object MER-2
 destination blender MER-3, MER-5, MER-28, MER-42
 initializing MER-36
 destination-blender setup functions MER-57
 destination-blending functions MER-5
 developing an application
 process of MER-3
 diffuse colo MER-27
 diffuse color MER-27
 draw function MER-5, MER-11
 draw functions MER-4
 drawing functions, see draw functions MER-4

E

engine
 rendering MER-1
 rendering, see also rendering engine MER-2
 example
 using lighting data MER-29
 using Mercury geometry MER-14

F

F = CS MER-30, MER-31
 faceted cube MER-16
 constructing MER-14
 faceted geometry MER-16
 faceted objects MER-13
 fan commands MER-12
 fans MER-12, MER-17
 fcameraskewmatrix transform MER-11
 fcamskewmatrix MER-12
 fcamax MER-12
 fcamy MER-12
 fcamaz MER-12
 Feiner, Computer Graphics Principles and Practice MER-viii

fields
 POD MER-10
 filepod.c MER-37
 filepod.c file MER-26
 firstPod MER-2
 flags field
 in Pod data structure MER-22
 fog MER-28
 creating MER-24
 fog color MER-28
 fog lighting
 described MER-28
 fogcolor MER-12
 Foley, Computer Graphics Principles and Practice MER-viii
 foreshortening
 scene MER-32
 frame buffer
 clearing MER-42
 frame buffer geometry MER-30
 frame-buffering MER-3, MER-36
 frames
 creating and rendering MER-42
 frontcullFLAG MER-23
 frustrum
 view MER-31, MER-32
 frustum MER-30
 fscreenheight MER-12
 fscreenwidth MER-12
 functions
 categories of MER-4
 core, see core functions MER-4
 destination-blending, see destination-blending funtions MER-5
 draw, see draw functions MER-4
 light, see light functions MER-5
 matrix 61, 61–114
 setup, see setup functions MER-5
 texture-blending, see texture-blending funtions MER-5
 utility, see utility functions MER-5
 vector 115–141
 fwclose MER-12
 fwfar MER-12

G

- game
 - creating MER-36
 - preparing to write
 - Mercury application
 - writing MER-37
- game code MER-3
 - writing MER-43
- game play MER-13
- gCornerPod MER-2
- gCornerPod object
 - definition of MER-3
- gDir MER-26
- geometries MER-2
- geometry
 - 3D MER-13
 - frame buffer MER-30
 - used by Mercury MER-13
- geometry in a 3D scene MER-2
- geometry model MER-13
- Getting Started With 3DO M2 Release 2.0 MER-viii
- gModelLightList MER-25
- Gouraud-interpolated colors MER-57
- gPoint MER-26
- Graphics Framework MER-38
- Graphics Pipelin MER-38
- graphics pipeline MER-1
- Graphics State, see also GState MER-37
- graphicsenv.c MER-37
- GS_Create function MER-39
- GS_GetDestBuffer MER-45
- GS_Reserve MER-44, MER-45
- GS_SetDestBuffer MER-39
- GS_SetVidSignal MER-39
- GS_WaitIO MER-45
- GState functions MER-3, MER-36, MER-37
- GState object MER-3
 - creating MER-39
 - creating and initializing MER-38
 - defined MER-38
 - initializing MER-37
 - main job of MER-38

H

- h MER-5
- header
 - PodGeometry MER-16
- helloworld MER-1, MER-25
 - Pod structure used in MER-21
- helloworld program
 - camera operations in MER-31
 - described MER-2, MER-36
 - examined MER-35
 - examined <\$endtrange> MER-46
 - executing MER-37
 - operations performed in MER-37
- helloworld.c MER-37
- hithernocullFLAG MER-23
- how used MER-2
- Hughes, Computer Graphics Principles and Practice MER-viii

I

- initialization commands
 - dispatching MER-42
- initialization operations
 - for a Mercury application MER-37
- initialization routine
 - for light sources MER-11
- initialization stage
 - of lighting routines MER-28
- input
 - user MER-43
- instruction cache
 - PPC MER-6
- instruction caching MER-7
- intensity value MER-28
- internal procedures MER-1
- internal processing MER-9

L

- libmercury_setup MER-7
- libmercury_util MER-7
- libmercury1 MER-6
- libmercury2 MER-6, MER-11
- libmercury3 MER-6, MER-11
- libmercury4 MER-7
- libraries
 - Mercury MER-6

-
- library hierarchy MER-6
 - light functions MER-5
 - described MER-24, MER-51, MER-52
 - light list
 - defined MER-25
 - described MER-25
 - Light List (the pod->plights field) MER-24
 - LightDir structure MER-26
 - lighting MER-2, MER-3, MER-5, MER-24–MER-29
 - lighting code MER-28
 - lighting data
 - using MER-29
 - lighting functions
 - described MER-24
 - lighting functions, see light functions MER-5
 - lighting operations MER-24
 - PODs and MER-3
 - lighting routines
 - calling sequence of MER-27
 - two stages of MER-28
 - lighting structure MER-25
 - lighting structures
 - how they work MER-25
 - lighting vertices MER-13
 - LightPoint structure MER-26
 - light-source initialization routine MER-11
 - linked list of PODs MER-2
 - linking
 - order of MER-7
- M**
-
- M_BoundsTest MER-5
 - M_BoundsTest function MER-43
 - described MER-49
 - M_DB functions MER-5
 - M_DBFog MER-59
 - M_DBFog function MER-57
 - M_DBInit MER-5, MER-44, MER-45
 - M_DBInit function MER-43, MER-58
 - M_DBNoBlend MER-58, MER-59
 - M_DBNoBlend function MER-57
 - M_DBSpec MER-59
 - M_DBSpec function MER-58
 - M_DBTrans MER-58, MER-59
 - M_DBTrans function MER-58
 - M_Draw MER-4, MER-5, MER-10, MER-11, MER-12, MER-22, MER-23, MER-24, MER-25, MER-26, MER-45
 - M_Draw function
 - described MER-48
 - M_Draw... functions MER-4
 - M_Draw... functions MER-11
 - M_Draw.s file MER-10
 - M_Draw.z file MER-11
 - M_DrawDynLit MER-4, MER-5, MER-58, MER-59
 - M_DrawDynLit function
 - described MER-51
 - M_DrawDynLitTex MER-5, MER-58, MER-59
 - M_DrawDynLitTex function
 - described MER-51
 - M_DrawDynLitTrans MER-59
 - M_DrawDynLitTrans function
 - described MER-51
 - M_DrawPreLit MER-5, MER-59
 - M_DrawPreLit function
 - described MER-51
 - M_DrawPreLitTex MER-59
 - M_DrawPreLitTex function
 - described MER-51
 - M_DrawPreLitTrans MER-59
 - M_DrawPreLitTrans function
 - described MER-51
 - M_End MER-4, MER-45
 - M_End function
 - described MER-48
 - M_Init MER-4, MER-40
 - M_Init function MER-38
 - described MER-47
 - M_Light MER-5, MER-11
 - M_Light... functions MER-5
 - M_Light... functions MER-11
 - M_LightDir MER-24, MER-25
 - M_LightDir function
 - described MER-53
 - M_LightDirSpec MER-24
 - M_LightDirSpec function
 - described MER-55, MER-56
 - M_LightDirSpecTex MER-24
 - M_LightFog MER-5, MER-24
 - M_LightFog function
 - described MER-52
 - M_LightFogTrans MER-24
 - M_LightFogTrans function
 - described MER-55
 - M_LightPoint MER-24

- M_LightPoint function
 - described MER-53
- M_LoadPodTexture MER-5
- M_LoadPodTexture function
 - described MER-49, MER-50
- M_Matrix_Perspective MER-5
- M_PreLight MER-5
- M_Prelight MER-13
- M_Prelight function MER-38
 - described MER-49
- M_SetCamera MER-5
- M_SetCamera function MER-30, MER-41, MER-42
- M_SetupDynLit MER-58
- M_SetupDynLitFog MER-59
- M_SetupDynLitFogTex MER-59
- M_SetupDynLitFogTrans MER-59
- M_SetupDynLitSpecTex MER-59
- M_SetupDynLitTex MER-58
- M_SetupDynLitTrans MER-58
- M_SetupDynLitTransTex MER-58
- M_SetupPreLit MER-59
- M_SetupPreLitFog MER-59
- M_SetupPreLitFogTex MER-59
- M_SetupPreLitFogTrans MER-59
- M_SetupPreLitTex MER-59
- M_SetupPreLitTrans MER-59
- M_SetupPreLitTransTex MER-59
- M_Sort MER-4, MER-22, MER-23, MER-45
- M_Sort function
 - described MER-48
- M_TB functions MER-5
- M_TBFog MER-59
- M_TBFog function MER-57
- M_TBLitTex MER-58, MER-59
- M_TBLitTex function
 - described MER-57
- M_TBNoTex MER-58, MER-59
- M_TBNoTex function
 - described MER-57
- M_TBTex function
 - described MER-57
- M2 coordinate system
 - transforming geometry into MER-13
- M2 destination blender, see destination blender
- MER-3
- Ma * La + Me MER-27
- Macintosh MER-vii
- Macintosh operating system MER-vii
- Macintosh Quadra MER-vii
- magic bit MER-17
- makefile
 - bigcircle MER-7
- mapping coordinates MER-13
- Material MER-21
- Material (the pod->pmaterial field) MER-24
- Material structure MER-25
 - defined MER-6
 - described MER-26
 - properties of MER-27
- materials
 - properties of MER-2
- Matrix MER-21, 63
- matrix
 - camera MER-31, MER-43
 - screen skew MER-43
 - skew MER-5, MER-30, MER-31
- Matrix (the pod->pmatrix field) MER-23
- matrix calls 61, 61-114
- Matrix fcamskewmatrix MER-12
- Matrix functions MER-5
- matrix functions 61, 61-114
- Matrix structure MER-31
 - defined MER-40, MER-41
 - example of MER-41
- Matrix_ 61, 64
- Matrix_BillboardX 61, 65
- Matrix_BillboardY 61, 66
- Matrix_BillboardZ 61, 67
- Matrix_Construct 61, 68
- Matrix_Copy 61, 69
- Matrix_CopyOrientation 61, 70
- Matrix_CopyTranslation 61, 71
- Matrix_Destruct 61, 72
- Matrix_FullInvert 61, 73
- Matrix_GetBank 61, 74
- Matrix_GetElevation 61, 75
- Matrix_GetHeading 61, 76
- Matrix_GetpTranslation 61, 78
- Matrix_GetTranslation 61, 77
- Matrix_Identity 62, 79
- Matrix_Invert 62, 80
- Matrix_LookAt MER-31, 62, 81
- Matrix_LookAt function MER-42
- Matrix_Move 62, 82
- Matrix_MoveByVector 62, 83
- Matrix_Mult 62, 84
- Matrix_Multiply 62, 85

Matrix_MultiplyOrientation 62, 86
 Matrix_MultOrientation 62, 87
 Matrix_Normalize 62, 88
 Matrix_Perspective MER-5
 Matrix_Perspective function MER-30, MER-33,
 MER-38, MER-41, MER-42
 Matrix_Print 62, 89
 Matrix_Rotate 62, 90
 Matrix_RotateLocal 62, 91
 Matrix_RotateX 62, 92
 Matrix_RotateXLocal 62, 93
 Matrix_RotateY 62, 94
 Matrix_RotateYLocal 62, 95
 Matrix_RotateZ 62, 96
 Matrix_RotateZLocal 62, 97
 Matrix_Scale 62, 98
 Matrix_ScaleByVector 62, 99
 Matrix_ScaleLocal 62, 100
 Matrix_ScaleLocalByVector 62, 101
 Matrix_SetTranslation 62, 102
 Matrix_SetTranslationByVector MER-31, 103
 Matrix_SetTranslationByVector function MER-42
 Matrix_Translate 62, 104
 Matrix_TranslateByVector 62, 105
 Matrix_Turn 62, 106
 Matrix_TurnLocal 63, 107
 Matrix_TurnX 63, 108
 Matrix_TurnXLocal 63, 109
 Matrix_TurnY 63, 110
 Matrix_TurnYLocal 63, 111
 Matrix_TurnZ 63, 112
 Matrix_TurnZLocal 63, 113
 Matrix_Zero 63, 114
 Mercury
 creating a game with MER-36
 described MER-1
 design of MER-1
 extending MER-1
 initializing MER-3, MER-36
 introducing MER-1–MER-34
 speed of MER-1
 understanding MER-2
 using MER-35–MER-46
 Mercury API MER-4
 Mercury application
 anatomy of MER-36–MER-46
 developing MER-3

Mercury functions
 and GState functions MER-36
 categories of MER-4
 Mercury geometry MER-13
 Mercury libraries MER-6
 Mercury rendering engine MER-1
 Mercury structures MER-6
 merge-sort technique MER-10
 model
 drawing MER-44
 geometry MER-13
 models
 drawing MER-43
 ModifyGraphicsItemVA MER-45

N

negative z axis MER-30
 Neider, Open GL Programming Guide MER-viii
 NEWS (startnewstripFLAG) MER-20
 Next-Pod Pointer (the pod->pnext field) MER-23
 nocullFLAG MER-23
 normals
 in faceted geometry MER-16

O

object descriptors MER-2
 object descriptors, see also PODs MER-2
 Open GL MER-viii
 Open GL Programming Guide MER-viii
 OpenGL Architecture Review Board MER-viii
 operating system
 Macintosh MER-vii

P

page-flippin MER-3, MER-36
 pcase MER-10
 PCLK flag MER-20
 per vertex stage
 of lighting routines MER-28
 per-frame operations MER-42
 perspective MER-41
 perspective transformation MER-32
 per-vertex functions MER-24
 PFAN flag MER-20
 pgeometry MER-10, MER-23
 PIP MER-11

- pipeline MER-1, MER-9
 - graphics MER-1
 - pipeline object descriptor, see also POD MER-2
 - Pipeline Object Descriptors (PODs) MER-2
 - Pipeline Object Descriptors, see also POD MER-2
 - pipeline object descriptors, see also PODs MER-2
 - plane
 - projection MER-32
 - plights MER-10, MER-24
 - plights pointer MER-27
 - pmaterial MER-24
 - pmaterials MER-26
 - pmatrix MER-10, MER-23
 - pmatrix transform MER-11
 - pnext MER-23
 - POD MER-2
 - described MER-2
 - Pod data structure
 - benefits of MER-21
 - defined MER-6
 - flags used in MER-22
 - POD fields MER-10
 - POD geometry MER-21
 - Pod Geometry the (pod->pgeometry field) MER-23
 - POD initialization stage MER-10
 - defined MER-10
 - described MER-10
 - POD object, see POD MER-3
 - Pod structure MER-21–MER-24
 - and PodGeometry structure MER-21
 - definition of MER-21
 - described MER-21
 - example of MER-21
 - in helloworld program MER-21
 - Pod Texture (the pod->ptexture field) MER-23
 - POD transforms MER-3
 - POD, see also pipeline object descriptor MER-2
 - pod->pcase field MER-23
 - pod->pgeometry field MER-23
 - pod->plight MER-24
 - pod->pmaterial MER-24
 - pod->pmatrix field MER-23
 - pod->pnext field MER-23
 - pod->ptexture field MER-23
 - pod->puserdata MER-24
 - PodGeometry (u, v) coordinates MER-21
 - PodGeometry data structur MER-13
 - PodGeometry flags MER-20
 - PodGeometry header MER-14
 - definition of MER-16
 - PodGeometry object
 - definition of MER-14
 - parts of MER-16
 - PodGeometry structure MER-14–MER-21
 - advantages of MER-13
 - constructing an object with MER-14
 - defined MER-6
 - example of MER-14
 - vertices of MER-16
 - PODs
 - converting to a command list MER-2
 - displaying MER-43
 - importance of MER-3
 - lighting MER-24
 - providing textures for MER-33
 - PODs, see also Pipeline Object Descriptors MER-2
 - point
 - transforming in world coordinates MER-31
 - Power Macintosh MER-vii
 - Power PC (PPC) 602 CPU MER-1
 - Power PC instruction cache MER-6
 - PPC instruction cache MER-6
 - prevclockwiseFLAG MER-20
 - procedures
 - internal MER-1
 - processing
 - internal MER-9
 - Program_Begin function
 - in helloworld program MER-43
 - projection plane MER-32
 - ptexture MER-10, MER-23
 - puserdata MER-10, MER-24
 - pyramid
 - view MER-33
-
- ## Q
-
- Quadra MER-vii
-
- ## R
-
- rendering MER-2
 - rendering engine MER-1, MER-2
 - described MER-1
 - rendering performance MER-3
 - requirements
 - system MER-vii

S

S = PT MER-32
 samecaseFLAG MER-22
 samecasePFLAG MER-10, MER-11
 sametextureFLAG MER-22
 sametexturePFLAG MER-10, MER-11
 scene
 created in helloworld MER-37
 scene foreshortening MER-32
 screen
 clearing MER-42
 screen skew matrix MER-43
 selecttextureFLAG MER-20
 Setup Case (the pod->pcase field) MER-23
 setup functions MER-5
 case MER-58
 destination-blender MER-57
 texture-blender MER-57
 shared vertices
 PodGeometry MER-16
 shine value MER-27
 skew matri MER-30
 skew matrix MER-5, MER-31, MER-41
 camera MER-41
 constructing MER-32
 created by Matrix_Perspective MER-30
 formula for constructing MER-30
 how it works MER-31
 screen MER-43
 skew matrix with perspective (camera skew matrix) MER-41
 SoftSpot function
 described MER-54
 sort code MER-10
 sort stage
 defined MER-9
 described MER-10
 specular color MER-27
 specular expone MER-27
 specular ligh MER-27
 specularFLAG MER-22, MER-27
 srcaddr MER-12
 srcaddr field MER-36
 stack MER-1
 stages of processing MER-9
 startnewstripFLAG MER-20
 strip commands MER-12

strip fa MER-16
 strip fan MER-17
 strips MER-12, MER-17
 STXT (selecttextureFLAG) MER-20
 summary
 Chapter 1 MER-34
 Chapter 2 MER-46
 System 7.1 MER-vii
 system requirements MER-vii

T

t PIP tabl MER-11
 texture blender MER-5, MER-28
 texture coordinates MER-12
 texture information
 accessing MER-33
 texture intensity MER-28
 texture load routin MER-11
 texture ma MER-28
 texture mapping MER-11
 texture numbers MER-17
 texture RAM (TRAM) MER-33
 texture-blender setup functions MER-57
 texture-blending functions MER-5
 texture-map (u, v) coordinates MER-16
 texture-mapping MER-33
 textures MER-2, MER-3, MER-21
 texturing MER-2
 total transform MER-11
 TRAM MER-11, MER-33
 loading MER-33
 TRAM, see also texture RAM MER-33
 transformation MER-2
 camera MER-32
 in skew matrix formula MER-30
 perspective MER-32
 transformation matrices MER-2
 transformations MER-13
 updating MER-13
 transforming vertices MER-13
 transforms
 POD, see POD transforms MER-3
 transparency MER-42
 transparency with fog MER-28
 triangle assembly stage MER-10, MER-11
 defined MER-10
 Triangle Engine MER-11, MER-38

triangles MER-17, MER-20
 clockwise MER-17, MER-20
typographical conventions MER-viii

U

UCoordColor valu MER-28
User Data (the pod->puserdata field) MER-24
user input
 handling MER-43
usercheckedclipFLAG MER-22, MER-23
utility functions MER-5
 described MER-49

V

van Dam, Computer Graphics Principles and Practice
 MER-viii
vector calls 115–141
Vector functions MER-5
vector functions 115–141
Vector3D_Add 115, 117
Vector3D_Average 115, 118
Vector3D_Compare 115, 119
Vector3D_CompareFuzzy 115, 120
Vector3D_Construct 115, 121
Vector3D_Copy 115, 122
Vector3D_Cross 115, 123
Vector3D_Destruct 115, 124
Vector3D_Dot 115, 125
Vector3D_GetX 115, 126
Vector3D_GetY 115, 127
Vector3D_GetZ 115, 128
Vector3D_Length 115, 129
Vector3D_Maximum 115, 130
Vector3D_Minimum 115, 131
Vector3D_Multiply 115, 132
Vector3D_MultiplyByMatrix 115, 133
Vector3D_Negate 115, 134
Vector3D_Normalize 115, 135
Vector3D_OrientateByMatrix 116, 136
Vector3D_Print 116, 137
Vector3D_Scale 116, 138
Vector3D_Set 116, 139
Vector3D_Subtract 116, 140
Vector3D_Zero MER-31, 116, 141
Vector3D_Zero function MER-42

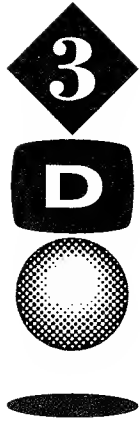
vertex indices MER-17
 PodGeometry MER-16
vertex pipeline stage MER-10
 defined MER-10
vertices MER-13
 clipping MER-12
 in PodGeometry structures MER-16
 lighting MER-11, MER-13
 of PodGeometry structure MER-16
 PodGeometry MER-16
 processing of MER-11
 transforming MER-13
vertices in PodGeometry structure MER-16
view frustrum MER-32
view frustum MER-31
view pyramid
 pointer to MER-31
view pyramid data MER-33
ViewPyramid object MER-42

W

Woo, Open GL Programming Guide MER-viii
world coordinate system MER-30
 mapping MER-13
world coordinates
 transforming a point in MER-31
world space
 camera matrix in MER-31

Z

z axis MER-30
Z-buffers
 setting MER-42



3DO M2 Command List Toolkit

Version 2.7 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

1 The Command List Toolkit

Introduction.....	CLT-2
Vertex Instructions	CLT-5
State Control Instructions.....	CLT-7
Flow Control Instructions	CLT-8
Command List Snippets	CLT-10

2 Graphics State (GState)

Introduction.....	CLT-12
Creating and Initializing a GState.....	CLT-12
Using a GState.....	CLT-13
Synchronizing the Triangle Engine to Video Signals through GState	CLT-14
Cleanup Routines	CLT-15
Combining CLT, the 2D and 3D Pipelines and Frameworks, and the Font Folio	CLT-15

3 Textures

Introduction.....	CLT-18
Texture Mapping	CLT-19
Using Filtering in Texture-Mapping Operations	CLT-19
Replicating Textures	CLT-19
Levels of Detail (LODs)	CLT-20
Using Mipmaps	CLT-21
Mipmap Filtering	CLT-21

How M2 Mipmap Filtering Works	CLT-22
Using Mipmap Filtering.....	CLT-24
Nearest (Point) Filtering	CLT-24
Linear Filtering	CLT-24
Bi-linear Filtering	CLT-26
Quasi Tri-linear Filtering	CLT-26
Perspective Correction.....	CLT-27
Computing the U and V Coordinates for Texture Mapping	CLT-27
Using Perspective Off Mode in 2D Operations	CLT-27
How Textures Are Stored in Memory.....	CLT-28
Texture Formats.....	CLT-29
How Texels Are Stored in Memory	CLT-29
How Uncompressed Texels are Stored	CLT-30
How Compressed Texels are Stored	CLT-30
Special Settings in the Control Byte	CLT-31
PIP Tables	CLT-32
How M2 Interprets Texel Data	CLT-33
How PIP Tables Work	CLT-33
Texture Application Blending	CLT-34
Multiplying Color and Alpha Components	CLT-34
Application Modes	CLT-35
Texture Mapping Step-by-Step	CLT-35
Loading Textures	CLT-36
Loading Textures by Using MMDMA	CLT-37
Loading Uncompressed Textures	CLT-37
Compressed Texture Load	CLT-39
Texture Mapper Pipeline Register Definitions.....	CLT-40
Command Registers, Fields, and Macros	CLT-40
Texture Generation Registers	CLT-42
Texture Loader Register Definitions	CLT-51

4

Destination Blender

Pixel Discards.....	CLT-60
Pixel Blending.....	CLT-60
Pixel Scaling	CLT-61
Source Blending	CLT-61
Dithering.....	CLT-62
Z-buffering.....	CLT-62

Z-banding	CLT-62
Window Clipping	CLT-63
Destination Blender Register Definitions	CLT-63
Register Memory Map	CLT-77

A

Register Setups

Z-Buffering	CLT-81
Dithering	CLT-82
Setting Up the Destination Blender Source Buffer	CLT-82
Transparencies	CLT-82
Texturing	CLT-83
Perspective	CLT-84
Sprites	CLT-84
Lighting	CLT-84
Tiling a Texture	CLT-86
Load an Uncompressed Texture	CLT-86
Loading a PIP Table	CLT-86

B

Sample Code

C

Function Calls

GState	CLT-94
Triangle Engine Command List Toolkit	CLT-95
Globals	CLT-95

List of Figures

Figure 1-1 Triangle Engine data flow.....	CLT-2
Figure 1-2 Texture Mapper data flow.....	CLT-3
Figure 1-3 Destination Blender data flow.....	CLT-4
Figure 1-4 Triangle strip.....	CLT-6
Figure 3-1 Texture Coordinates.....	CLT-18
Figure 3-2 The texture-mapping process.....	CLT-19
Figure 3-3 Mipmaps.....	CLT-20
Figure 3-4 Tradeoffs in filtering operations.....	CLT-23
Figure 3-5 Bi-linear filtering.....	CLT-26
Figure 3-6 Storage format of an uncompressed texel.....	CLT-30
Figure 3-7 Storage format of a ompressed texel.....	CLT-30
Figure 3-8 Bits in the control byte.....	CLT-30
Figure 3-9 How textures and texels work with PIP tables.....	CLT-32
Figure 3-10 Rendering a ghost.....	CLT-33
Figure 3-11 Bit packing of a Color.....	CLT-34
Figure 3-12 Placing Load Rectangles inside a texture.....	CLT-36
Figure 3-13 An array of uncompressed texels ready for loading.....	CLT-37
Figure 3-14 Uncompressed Loader Setup.....	CLT-38
Figure 3-15 TxtLdSrcType01.....	CLT-52
Figure 3-16 TxtLdSrcType23.....	CLT-53
Figure 3-17 TxtExpType.....	CLT-53
Figure E-1 Dithering matrix.....	CLT-82

List of Tables

Table 3-1 Texture Filtering Modes CLT-22

Table 3-2 Registers for specifying LOD Regions CLT-24

Table 3-3 Texel Types..... CLT-29

Table 3-4 Color, Alpha, and SSB Values that Can Be Assigned to a Texel CLT-29

Table 3-5 Register Memory Map CLT-41

Preface

About This Book

This book describes the Command List Toolkit and its relationship to the M2's 3D hardware rendering engine.

About the Audience

This manual is written for title developers. The information presented and the level of description is based on the assumption that you are familiar with both the C programming language, and three-dimensional graphics.

How The Command List Toolkit Programmer's Guide Is Organized

Chapter 1, The Command List Toolkit –Provides a brief overview of the M2 3D hardware rendering engine, an introduction to the Command List Toolkit, and descriptions of the associated macros.

Chapter 2, The Graphics State (GState) –Describes the GState object and how it manages the command list buffers.

Chapter 3, Textures –Provides an overview of the M2's texture mapping logic, as well as descriptions of all of the applicable registers.

Chapter 4, Destination Blender –Provides an overview of the M2's destination blending logic, as well as descriptions of all of the applicable registers.

Appendix A, Register Setups –Provides the register setups for a variety of operations.

Appendix B, Sample Code –Provides a sample program.

Appendix C, Function Calls –Describes each of the Command List Toolkit function calls.

Related Documentation

The following documents are recommended as a source of additional information:

- ◆ Foley, van Dam, Feiner, and Hughes, *Computer Graphics Principles and Practice*, Addison-Wesley, 1990
- ◆ 3DO M2 Graphics Programmer's Guide

The Command List Toolkit

The M2's 3D hardware rendering engine (the Triangle Engine or TE) renders graphics by reading blocks of data, called *command lists*, from memory and executing the instructions found in these blocks. The Command List Toolkit (CLT) is a collection of routines and macros that allow you to build these command lists easily and efficiently. Applications use the CLT to build commands for the Triangle Engine. These commands are built up into command list buffers, then handed off to the Triangle Engine's device driver.

The TE executes the commands, found in the command list buffers, asynchronously. So, while the Triangle Engine works on one block of command lists, the CPU is free to begin work on preparing the next set of Triangle Engine commands.

The main job of the CLT is to assign a set of names to all of the Triangle Engine registers and commands. These names are provided to the user through a header file. In addition, several shortcuts for using these names are provided in the form of C preprocessor macros.

This chapter describes the macros provided by the CLT. Your application can use these macros to communicate, at the most rudimentary level, with the Triangle Engine.

This chapter contains the following topics:

Topic	Page Number
Flow Control Instructions	8
Vertex Instructions	8
State Control Instructions	8

Topic	Page Number
Flow Control Instructions	8
Command List Snippets	8

Introduction

Before we talk about the Command List Toolkit, let's take a brief look at how the Triangle Engine works. The TE is primarily composed of two groups of logic: the Texture Mapper and the Destination Blender. illustrates the data flow through the Triangle Engine. See Chapters 3 and 4 for more information about the Texture Mapper and the Destination Blender.

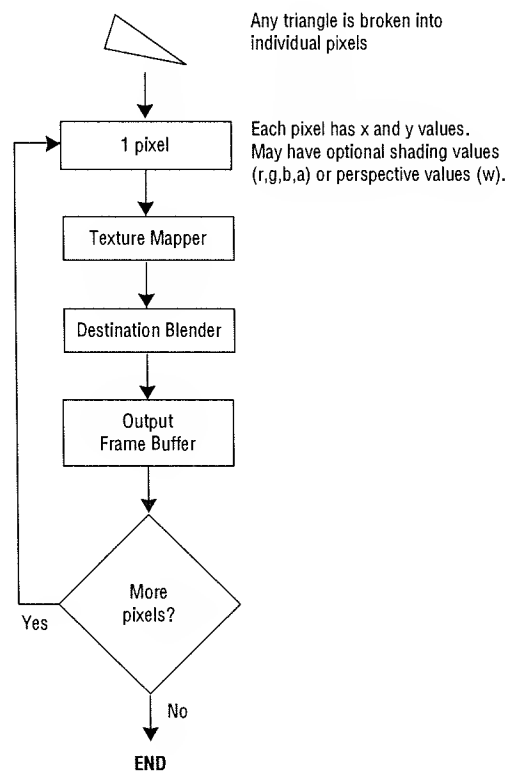


Figure 1-1 Triangle Engine data flow.

The TE renders triangles by first breaking them down into individual pixels. Each pixel has a set of *xy* coordinates that identify it, and may have optional values that define its shading or perspective.

The pixel goes first to the Texture Mapper. The texture mapper logic can render portions of a 2D image onto a 3D triangle, and it is here where texture and color are applied to the pixel (see Figure 1-2).

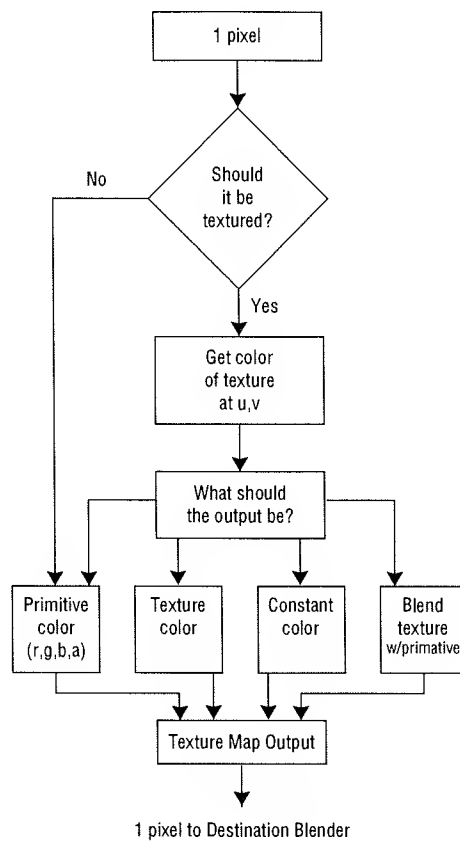


Figure 1-2 *Texture Mapper data flow.*

Next stop is the Destination Blender. The destination blender logic takes the pixel's texture information, and combines (blends) it with data from a number of sources. These sources can include the current Source frame buffer, a variety of constants and the control registers (see Figure 1-3).

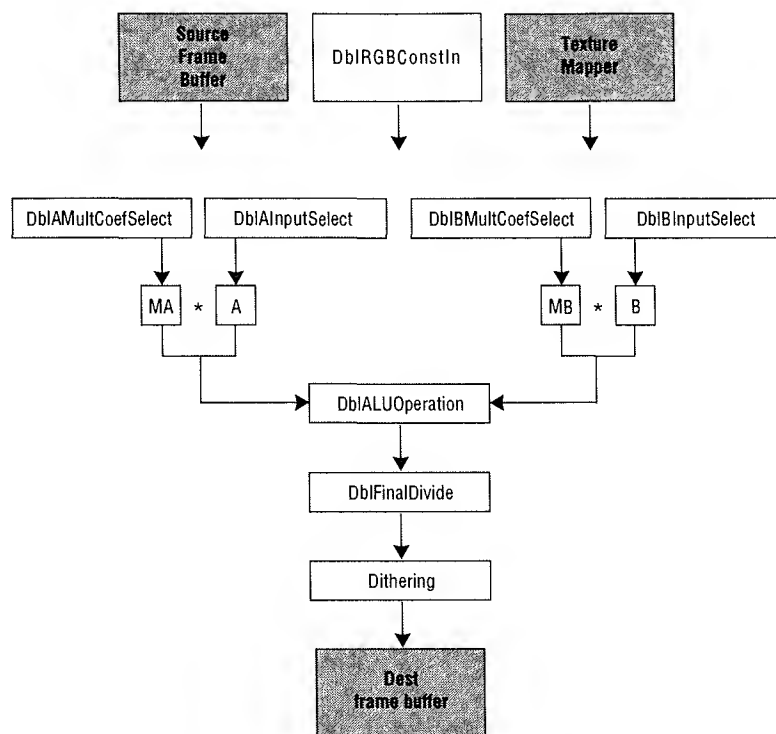


Figure 1-3 Destination Blender data flow.

The Triangle Engine works in a *deferred mode*, meaning that it executes commands from a buffer, rather than responding immediately to each of the commands that you issue. The normal flow of these commands usually looks similar to the list below:

1. Provide information about texture's format, location in memory, etc.
2. Load the texture.
3. Provide information about the texture's PIP (Pen Index Palette, or color table).
4. Load the PIP.
5. Set up the other hardware registers, such as the Destination Blend unit.
6. Provide vertices of one or more triangle strips or fans, to be rendered using the parameters described in Steps 1 - 5, above.
7. Issue a SYNC instruction. Refer to "Flow Control Instructions," later in this chapter, for more information about the SYNC instruction.
8. Return to Step 1.

The only step in the process that actually causes pixels to be rendered is Step 6. The other steps describe how those pixels will be rendered. Note that the SYNC instruction is needed after a collection of vertex instructions have been issued to the Triangle Engine, but it is described in more detail later.

Vertex Instructions

Since vertex instructions are what actually cause pixels to be rendered, we'll start by describing the CLT macros that create these instructions. For now, let's assume that the Texture Mapper, Destination Blender, and the other setup portions of the Triangle Engine are all set to allow these instructions to render correctly.

The general form of vertex instructions is one header word, followed by three or more vertices. The header word describes what the format of the vertices will be, a count of the vertices that follow, and also tells the Triangle Engine whether the vertices describe a triangle strip, or a triangle fan. Vertices contain x and y data, and may also contain RGBA color data, uv texture mapping data, and a $1/w$ perspective correction value (also used for depth buffering). The actual contents of a vertex instruction depends on the macro you choose to use.

The macro to create a vertex header instruction is of the form:

```
CLT_TRIANGLE (pp, stripfan, perspective, texture, shading, count)
```

`pp` – the address of a pointer to the current insertion point in a command list buffer. See the section on GState for more details about this command list buffer pointer. The address of the pointer is necessary, because CLT updates the pointer as it adds instructions to the command list buffer.

Note: See Chapter 2 for more details about the command list buffer pointer.

`stripfan` – an integer stating whether the vertices that follow should be interpreted as a triangle strip or a triangle fan. The constants `RC_STRIP` and `RC_FAN` should be used for this parameter.

`perspective` – a bit stating whether the vertices that follow contain a $1/w$ perspective correction value.

`texture` – a bit stating whether the vertices that follow contain U and V texture coordinates.

`shading` – a bit stating whether the vertices that follow contain R, G, B, and A (alpha) color data.

`count` – an integer count of how many vertices follow this header word. It is important to note that the Triangle Engine can work more efficiently on longer strips and fans than on shorter ones. If optimal rendering speed is desired, the average number of triangles per strip or fan should be 10 or more.

There are several forms of the CLT vertex macros. Each form covers one of the various triangle formats described by the header word. Some examples follow:

```
CLT_Vertex(pp, x, y)
CLT_VertexRgbaW(pp, x, y, r, g, b, a, w)
CLT_VertexRgbaUvW(pp, x, y, r, g, b, a, u, v, w)
```

`pp` – the address of a pointer to the current insertion point in a command list buffer. As with all CLT macros that take `pp`, it is updated as data is written to the buffer.

All of the remaining values for the `CLT_Vertex` instructions are represented as IEEE 754 standard 32-bit floating point numbers.

x , y – the x and y coordinates of the vertex of a triangle. They should be in the range of 0 to bitmap width, and 0 to bitmap height, respectively. Negative numbers are not allowed for x and y , so care has to be taken to clip triangles against the top and left edges of a bitmap. Triangles that extend beyond the right or bottom edges of a bitmap are clipped by the Triangle Engine, if they are within the range of 0.0 to 2048.0.

r , g , b , a – represent the red, green, blue, and alpha components for shading a triangle. This data might be different at each vertex, to allow for Gouraud shading of the triangles, or it can be the same at each vertex, in order to flat shade the triangles. The values should be in the range of 0.0 to 1.0 for each of these components.

u , v – represent the texture coordinates that should be applied to a triangle. The values should be in the range of 0 to texture width, and 0 to 1024, respectively. When perspective correction of the textures is desired, u and v should actually be sent to the Triangle Engine as u/w and v/w . Some textures can be tiled, meaning that its texel data repeats in the x and/or y directions. Note that if u and v exceed the width of the texture, the clamping mask is applied.

$1/w$ – a perspective correction factor. As a result, the range for the w value is $[0,1)$. When $w = 0.0$, a triangle is represented as being infinitely far away. As w approaches 1.0, the perspective correction applied to it becomes less and less, until eventually there is no correction done on the triangles.

Below is a fragment of code that adds two triangle strips to a command list buffer. Again, this code assumes that the texture mapper and destination blender have been correctly set up, and that the variable `listPtr` is a valid pointer to the current insertion point in a command list buffer.

```
{
    ...
    CLT_Triangle( &listPtr, RC_STRIP, 0, 1, 0, 6 );
    CLT_VerTexRgba( &listPtr, 20.0, 20.0, 0.0, 0.0, 0.0, 1.0 );
    CLT_VerTexRgba( &listPtr, 23.0, 60.0, 0.1, 0.1, 0.1, 1.0 );
    CLT_VerTexRgba( &listPtr, 110.0, 10.0, 0.7, 0.7, 0.7, 1.0 );
    CLT_VerTexRgba( &listPtr, 95.0, 80.0, 0.7, 0.7, 0.7, 1.0 );
    CLT_VerTexRgba( &listPtr, 200.0, 20.0, 0.2, 0.2, 0.2, 1.0 );
    CLT_VerTexRgba( &listPtr, 190.0, 60.0, 0.14, 0.14, 0.14, 1.0 );
    ...
}
```

This piece of code outputs a strip comprised of six untextured, shaded triangles. The triangles start at roughly 20 pixels into the bitmap from the left, and move right until they are about 220 pixels in from the left edge of the bitmap. As they near the center, they get brighter.

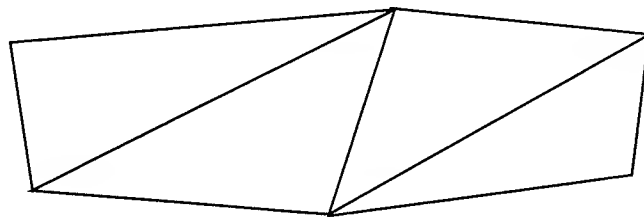


Figure 1-4 Triangle strip

State Control Instructions

The way in which a triangle is actually rendered in a bitmap is controlled by the state of the Triangle Engine. For example, the triangles in the previous sample code might have a texture blended with the shading, might be dithered, or might be rendered as partially translucent triangles. In this section, the methods for changing this state are described. The individual state-controlling registers are described in Chapters 3 and 4.

The state registers for the Triangle Engine are all 32-bits, although some features may only use some of the bits in a register, while others may be affected by data from more than one register. The Triangle Engine provides three ways of writing data to these registers, and so the CLT provides three methods as well:

```
CLT_WriteRegister( pp, reg, val )
CLT_SetRegister( pp, reg, val )
CLT_ClearRegister( pp, reg, val )
```

The first command, `CLT_WriteRegister`, stores the value `val` in the register `reg`. The second command, `CLT_SetRegister`, ORs the bits in `val` into the previous value of the register `reg`. The last command, `CLT_ClearRegister`, clears any bits in `reg` if the corresponding bits in `val` are set. For each of these calls, `pp` is the address of a pointer to the insertion point in a command list buffer.

These macros actually write a header word to the command list buffer, then the 32-bit quantity, `val`. This header word describes whether a Set, Clear, or Write should occur, at what address the change should occur, and how many consecutive registers are written. Each of the macros (`CLT_WriteRegister`, `CLT_SetRegister`, and `CLT_ClearRegister`) always write only one register at a time. If a user program wants to change the state of several registers at once, and these registers resided in consecutive addresses, a more efficient method would be to use:

```
CLT_WriteRegistersHeader( reg, cnt )
CLT_SetRegistersHeader( reg, cnt )
CLT_ClearRegistersHeader( reg, cnt )
```

Note that these CLT macros do not take `pp` as an argument, so user code has to look something like the example below:

```
{
    ...
    CLT_WriteRegister( pp, DBCONSTIN, 0xFFA01010 ); /* mostly red */
    *pp++ = CLT_WriteRegistersHeader( DBDITHERMATRIXA, 2 );
    *pp++ = 0xC0D12E3F; /* DBDITHERMATRIXA was just set */
    *pp++ = 0xE1D0302F; /* DBDITHERMATRIXB was just set */
    ...
}
```

Whenever these macros need to name a register, they can use one of the named register constants. In the CLT header file, all of the named registers are of the form `RA_<regName>`. For example, `RA_DBDITHERMATRIXA` is the register address of the top half of the destination blender's dither matrix.

Because some attributes might only use a few bits of one of the registers, the CLT header file also defines several bit shifts and masks that can be used to set just portions of a register. Each field within each register has been named in the form `FV_<regName>_<Attribute>`. For each of these field values, the shift and mask can be used to ensure that values do not write into other register attributes accidentally.

The following macros can be used in combination with the CLT_ClearRegister and CLT_SetRegister macros to only change a portion of register, so that the remaining fields within the register can remain intact.

```
CLT_Mask( reg, field )
CLT_Bits( reg, field, data )
```

The following code fragment sets only the red portion of the Destination Blender's BMultConstSSB0 register, to the value 0x40:

```
{
    ...
    CLT_ClearRegister( pp, DBBMULTCONSTSSB0,
    CLT_Mask(DBBMULTCONSTSSB0, RED ) );          /* Clear red channel */
    CLT_SetRegister( pp, DBBMULTCONSTSSB0,
    CLT_Bits(DBBMULTCONSTSSB0, RED, 0x40 ) ); /* OR in 0x40 for red */
    ...
}
```

For some of the field values, a constant makes more sense than a numeric value. These constants have also been defined in the CLT's header file, along with some macros to use them. The CLT_Const macro can be used to retrieve one of these constants to provide a value for the data field of the CLT_Bits macro. The following example illustrates one way to use the CLT_Const macro. It sets the InterFilter of the Texture Mapper to TriLinear filtering.

```
{
    ...
    CLT_ClearRegister( pp, TXTADDRCTL,
    CLT_Mask( TXTADDRCTL, INTERFILTER ) );
    CLT_SetRegister( pp, TXTADDRCTL,
    CLT_Bits( TXTADDRCTL, INTERFILTER,
    CLT_Const( TXTADDRCTL, INTERFILTER, TRILINEAR ) ) );
    ...
}
```

For each of the state registers in the Texture Mapper and Destination Blender, the CLT header file also provides macros that take all of the necessary field values as arguments, and outputs the correct CLT_WriteRegister command. Using these simpler macros, in cases where all of the data in a register needs to be set at once, can make the code appear cleaner.

Flow Control Instructions

The Triangle Engine has some additional instructions that affect its data flow. These instructions, collectively known as the *flow control instructions*, start and stop the Triangle Engine, cause it to jump to a new place within a command list buffer, and can instruct it to actually load a PIP or texture from main RAM. The flow control instructions can use the same macros as the state control registers (described above) or any of the following macros:

```
CLT_Sync(pp)
CLT_Pause(pp)
CLT_Interrupt(pp, ival)
CLT_JumpRelative(pp, address)
CLT_JumpAbsolute(pp, address)
CLT_TxLoad(pp)
```

Caution: It is important to understand the use of these registers, because using them incorrectly can cause the Triangle Engine to crash.

The SYNC instruction tells the Triangle Engine to flush all of its internal pipelines. This command **must** be used to separate one or more consecutive triangle strips and/or triangle fans from any state control instructions that may follow the vertices. Failure to use the SYNC instruction, after vertices have been rendered, can lead to Triangle Engine crashes.

The PAUSE instruction works like the SYNC instruction, except that it also causes the Triangle Engine to stop at the word after the PAUSE. The TE remains stopped until it is restarted at some later time by the CPU. When a PAUSE is encountered, an interrupt can be generated if the Triangle Engine was set to cause an interrupt when it reached the end of a list. The command to tell the Triangle Engine to begin, once a PAUSE has been encountered, is a command that must be executed by the Triangle Engine Device Driver. Each command list buffer normally contains a PAUSE instruction after the last state control or vertex instruction in the buffer.

The INT (interrupt) instruction tells the Triangle Engine to cause an interrupt for the CPU. This should not be used by applications, since the Triangle Engine device driver currently does not handle special interrupts.

The JR (jump relative) instruction causes the Triangle Engine to jump to a different address within a command list to fetch instructions. The relative offset from the jump instruction (a 2's complement number), should first be loaded into the RA_DCNTLDATA register by using the CLT_WriteRegister macro. Note that this is normally not needed by applications.

The JA (jump absolute) instruction causes the Triangle Engine to jump to an absolute address to fetch further instructions. The address should first be loaded into the RA_DCNTLDATA register by using the CLT_WriteRegister macro. Note that this is normally not needed by applications.

The TLD (texture load) instruction is used throughout a command list buffer whenever it becomes necessary to load a new texture into the Triangle Engine's Texture RAM (TRAM). To load a texture successfully, it is normally necessary to set up several registers in the Texture Mapper first, such as the load address, etc. Once all of these registers are set up, you issue a TLD command before any vertex instructions attempt to use the new texture. Note that it is not necessary to SYNC immediately before or after a TLD instruction, unless it happens to be an instruction that follows a vertex instruction.

Command List Snippets

So far, this document has described the CLT macros as tools for writing into a command list buffer. There are many cases, however, where it is useful to build a block of Triangle Engine commands once, then re-use the pre-built data several times. One way to prevent your code from having to re-compute these macros, each time a repeated set of commands needs to be applied, is to write the commands to a *Command List Snippet*, or `CltSnippet` data structure. These snippets can be dynamically allocated and filled once with commands. Then, only a simple `memcpy` is necessary to place the data into a command list buffer. In the M2 Graphics Pipeline code, several command list snippets are saved off and re-used, to help save computing cycles. For example, one command list snippet is provided that disables texturing. Whenever a non-textured object follows a textured object, one simple call copies data into the command list buffer to disable the Texture Mapper. A `CltSnippet` is defined as follows:

```
typedef struct {
    uint32* data;
    uint16 size;           /* Number of words used */
    uint16 allocated;     /* Number of words allocated */
} CltSnippet;
```

To allocate a new command list snippet, call:

```
int CLT_AllocSnippet(CltSnippet *s, uint32 numWords)
```

This routine allocates enough memory for the specified number of commands to be placed into this `CltSnippet`.

There may be cases when it is useful to create one `CltSnippet`, then re-use variations of it several times. To copy one snippet into another, call:

```
void CLT_CopySnippet(CltSnippet *dest, CltSnippet *src)
```

The `dest CltSnippet` must already have been allocated, and its data area must be at least as large as the `src CltSnippet`.

Once these snippets are built, the routine that copies a command list from the `CltSnippet` into a command list buffer is:

```
void CLT_CopySnippetData(uint32 **dest, CltSnippet *src)
```

No check is done to ensure that there is enough space remaining in the command list buffer for the whole snippet, so it is the responsibility of the application to do this before attempting to copy the data into the command list buffer.

Once a command list snippet is no longer needed, the memory allocated to it can be freed with a call to:

```
void CLT_FreeSnippet(CltSnippet *s)
```

Graphics State (GState)

The Graphics State (GState) object helps users manage the command list buffers that store the data used by the Triangle Engine to render graphics.

The main job of the GState is to provide an easy way of creating command list buffers, filling them with Triangle Engine instructions, and eventually sending them to the Triangle Engine device driver, so that the instructions can be executed. The GState is designed to give the user flexibility in how memory is used for command list buffers while keeping the overhead, both in memory and in CPU cycles, to a minimum.

This chapter describes how to use the GState to create command list buffers, and how to correctly use them so that user code, the 2D and 3D Pipelines and Frameworks, and the Font Folio, can all work together to render graphics through the Triangle Engine.

This chapter contains the following topics:

Topic	Page Number
Introduction	12
Creating and Initializing a GState	12
Using a GState	13
Synchronizing the Triangle Engine to Video Signals through GState	14
Cleanup Routines	15
Combining CLT, the 2D and 3D Pipelines and Frameworks, and the Font Folio	15

Introduction

The 2D Pipeline and Framework, the 3D Pipeline and Framework, the Command List Toolkit, and the Font Folio, all write commands to the command list buffers. Providing one standard way for all of these components to write to and send command list buffers results in the following benefits:

- ◆ Commands from all of the different sources can be intermingled throughout the process of rendering a scene
- ◆ Memory used for command list buffers can be shared
- ◆ Code used to manage command list buffers can be shared

The GState is a data structure and collection of routines that encapsulate command list buffers in a way that all of these libraries and folios can understand and use. Routines are provided to create command list buffers, determine where the current insertion point is in a command list buffer, and to send a command list buffer to the Triangle Engine device driver. For optimal performance, command lists should be double-buffered, so that while the Triangle Engine is executing the instructions in one buffer, the CPU can begin filling the second buffer with the next batch of commands. The GState supports double-buffered command lists.

A GState object looks like the following structure:

```
typedef struct GState {  
    Err      (*gs_SendList)(struct GState*);  
    CmdListP gs_EndList;  
    CmdListP gs_ListPtr;  
} GState;
```

In addition, there are several private fields which must be accessed through the interface routines provided.

Creating and Initializing a GState

Before command lists can be created, a GState needs to be created. To create a GState, an application should call:

```
GState* GS_Create(void);
```

Memory is allocated for a GState structure, and the structure is initialized to a default state. Users should *not* allocate a GState as a stack variable, because the structure contains many fields that are private. The next step is to allocate one or more command lists on a GState. The following routine allocates the specified number of lists, and assigns them to a GState:

```
Err GS_AllocLists(GState* g, uint32 numLists, uint32 listSize);
```

Note that `listSize` is specified in WORDS, not bytes. Command lists are just blocks of user memory, like any other blocks allocated with `AllocMem()` or `malloc()`.

The other setup necessary for a GState is to specify what bitmap the command lists should be rendered into, and optionally, what bitmap to use as a Z-buffer. These buffers are set and retrieved with the following calls:

```

Err GS_SetDestBuffer(GState* gs, Bitmap* bm);
Err GS_SetZBuffer(GState* gs, Bitmap* bm);
Bitmap* GS_GetDestBuffer(GState* gs);
Bitmap* GS_GetZBuffer(GState* gs);

```

When double-buffering frame buffer bitmaps, it is usually desirable to create *tear-free rendering*. To accomplish this, an application can associate a signal bit with a GState and the Graphics Folio. When the Graphics Folio finishes displaying a bitmap, it sends a signal to a task, stating that the previously-displayed bitmap is completely off the screen, and thus safe to render into. If a signal is allocated for use as a display signal in the Graphics Folio, it can be associated with a GState by calling:

```
Err GS_SetVidSignal(GState* gs);
```

If the signal is set on a GState, the GState waits before sending the first command list buffer for a bitmap.

Using a GState

Once a GState has been set up, data can be written to it's command list buffers by writing 32-bit words to a pointer to the current insertion point in the current command list buffer. The GState maintains such a pointer in the field `gs_ListPtr`. Most of the Command List Toolkit macros, that write data to a command list buffer, take the address of this field as one parameter, and update the `gs_ListPtr` as they write data to the buffer.

The 2D and 3D Pipelines already do their own checks to ensure that there is enough space in a command list buffer before issuing a block of commands. However, when using CLT, it is up to the application to ensure that there is enough room remaining in the buffer. The following call ensures that there is enough memory for the requested number of words of space:

```
void GS_Reserve(GState* gs, uint32 numWords);
```

If sufficient space isn't available, the current command list buffer is sent to the Triangle Engine, and the next available command list is made current. The `gs_ListPtr` field then points to the beginning of this new command list buffer. Note that `GS_Reserve()` will fail if an application requests more words than are actually available in an entire list. If this happens, the results are unpredictable. It is a good practice, while developing an application, to allocate plenty of space to the command list buffers. You can reduce the size of the buffers once the rest of the code has stabilized. At this point, you can make a good estimate of what the minimum buffer size needs to be. Note that `GS_Reserve()` does not actually advance the insertion point by any amount of space. It is up to user code, or CLT macros, to advance the insertion pointer as data is written to the buffer.

When `GS_Reserve()` does not find enough space available in the current command list buffer, it calls the function pointed to by the `gs_SendList` field. This field normally points to the following routine:

```
Err GS_SendList(GState* gs);
```

This routine can be called at any time by the application. It flushes the current command list buffer to the Triangle Engine. `GS_SendList()` sends a command list buffer to the Triangle Engine to be rendered. This routine can optionally wait for a video signal saying that it is safe to start rendering to a buffer, and will ensure that `gs_ListPtr` is pointing to an available command list buffer before

returning. `GS_SendList()` calls `SendIO` to send the command list buffer to the Triangle Engine device driver. The following routine can be used to wait for all current and pending `IORequests` to complete:

```
Err GS_WaitIO(GState* gs);
```

Normally, this routine is called before attempting to display a buffer in a double-buffering scheme, so that the application can ensure that Triangle Engine rendering is not visible to the end user.

If it becomes necessary to write your own routine to send a list to the Triangle Engine, the following few routines might be helpful:

```
CmdListP GS_GetCurListStart(GState* g);
```

This routine returns a pointer to the beginning of the current command list buffer. In order to send a list to the Triangle Engine device driver, it is necessary to specify the start of a command list buffer, the end of the buffer, and some frame buffer information. Look at the source for `GS_SendList()` for exact details on how to send a command list.

```
void GS_SetList(GState* g, uint32 idx);
```

`GS_SetList()` sets which command list to use for the given `GState`. The `idx` field is in the range of `0...(num cmd lists - 1)`.

```
uint32 GS_GetCmdListIndex(GState* g);
```

This routine returns the index of the command list currently in use. Normally, you would call `GS_SetList()` with `(this index + 1) % (num cmd lists)` after sending the current list.

Synchronizing the Triangle Engine to Video Signals through GState

As mentioned earlier, it is possible to set up a `GState` to provide tear-free double-buffering of frame buffers. The basic principle is: Before a bitmap can be used as an output buffer, the `GState` ensures that it is not currently being displayed on the screen. The Graphics Folio can send signals that, if used correctly, allow the `GState` to always write only to buffers that are not up on the display. In combination with the correct calls to the Display Folio, buffers are swapped onto the screen with no visible tearing. The basic steps to follow to achieve this tear-free double-buffering are:

- ◆ Allocate a signal bit by calling `AllocSignal()`.
- ◆ When a view is created, define this signal as the display signal. This is done by specifying the signal as the argument for the `VIEWTAG_DISPLAYSIGNAL` tag when calling `CreateItem` for the view.
- ◆ Create a `GState` by calling `GS_Create()`.
- ◆ Associate the signal with the `GState`. This is done by calling the routine:

```
Err GS_SetVidSignal(GState* gs, int32 signal);
```
- ◆ At the beginning of each frame to be displayed as a frame buffer, call:

```
Err GS_BeginFrame(GState* gs);
```

If rendering to a buffer that is not going to be immediately displayed, do not call `GS_BeginFrame()`. When `GS_SendList()` sees that `GS_BeginFrame()` has been called, and a signal is associated with a `GState`, then `GS_SendList()` waits

until that signal is activated by a call to `ModifyGraphicsItem()`. If a bitmap is going to be rendered into, and `ModifyGraphicsItem()` is not called to display it, the signal is never sent to the GState, and the application effectively hangs.

Cleanup Routines

To free the memory allocated to the command list buffers for a GState, call:

```
Err GS_FreeBuffers(GState* gs);
```

If, for some reason, it is necessary to re-allocate new command list buffers, `GS_AllocLists()` can be called again.

Once an application is completely finished using a GState, the GState can be freed by calling:

```
Err GS_Delete(GState* gs);
```

If `GS_FreeBuffers()` has not been called prior to `GS_Delete()`, it is called implicitly as the GState is freed in memory.

Combining CLT, the 2D and 3D Pipelines and Frameworks, and the Font Folio

The 2D and 3D Graphics Pipelines, and the Font Folio, are all built on top of GState. One benefit of this is that calls from these various sources can be intermingled throughout a frame. For example, if an application is using the 3D Pipeline to render some scenery for a flight simulator, and then going to use the Font Folio to display a status bar with altitude, air speed, etc., the application could set up the scene, and tell the 3D Pipeline to render it. Before the command lists are flushed, the application could then call the Font Folio routines to display the appropriate text. A final call to `GS_SendList()` would then render all of the commands from the Pipeline and the Font Folio into the bitmap at once.

Textures

In M2, a *texture* is a two-dimensional image that is used to modify the color or the transparency of a 3D object. A texture follows the surface that it is mapped to—often folding, bending, or wrapping around to follow the contours of a 3D object.

This chapter contains the following topics:

Topic	Page Number
Introduction	18
Texture Mapping	19
Mipmap Filtering	21
Using Mipmap Filtering	24
Perspective Correction	27
How Textures Are Stored in Memory	28
Texture Formats	29
How Texels Are Stored in Memory	29
PIP Tables	32
Texture Application Blending	34
Texture Mapping Step-by-Step	35
Loading Textures	36
Texture Mapper Pipeline Register Definitions	40

Introduction

A texture can be something as simple as a color (with or without various degrees of shading) or something as complex as (or more complex than) a tiled pattern, a newspaper page, a brick wall, or even a previously-rendered frame buffer. The texture pipeline is capable of processing from 1 pixel every 3 ticks to 2 pixels per tick, depending on the filtering being applied.

In the M2 system, textures are stored in arrays, and each element of the array in which a texture is stored is called a *texel*. Each texel can be up to 32 bits long. For more information about the structure of a texel, see "How Textures Are Stored in Memory."

The coordinate system for a texture is shown in Figure 3-1. By convention, the horizontal and vertical coordinates of a texture are expressed as U and V , as shown in the diagram. If U is a variable that is used to represent horizontal coordinates and V is a variable that is used to represent vertical coordinates, the point where $\langle U, V \rangle = \langle 0, 0 \rangle$ is at the top-left hand corner of the texture's mipmap, and the address of that point is considered the base address of the texture. This means that in the case of an 8-bit texture, the map is stored in memory with U increasing by one from one memory location to the next.

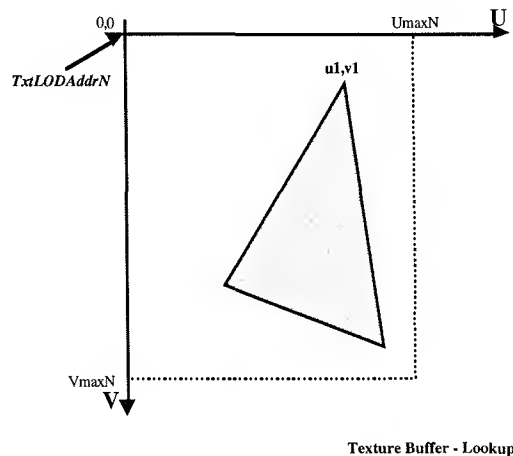


Figure 3-1 *Texture Coordinates*

Note: U and V coordinates must be in the range from 0 to 1023.

For more information about how M2 stores textures in memory, see "How Textures Are Stored in Memory." See Appendix A for the register setup used to turn texturing on and off.

Texture Mapping

Texture mapping is the process of mapping a texture onto a 3D object displayed on a screen. Texture mapping can be tricky because a texture is a 2D rectangular region that must often be mapped to a 3D non-planar surface with various kinds of irregular features. For example, Figure 3-2 shows how a texture might be mapped to a pyramid-shaped 3D object.

The texture shown in the rectangle on the left has been mapped to the pyramid shown on the right. Notice that the texture obtained from the texture on left is wrapped around to match the shape of the pyramid and is distorted when it is mapped to the pyramid.

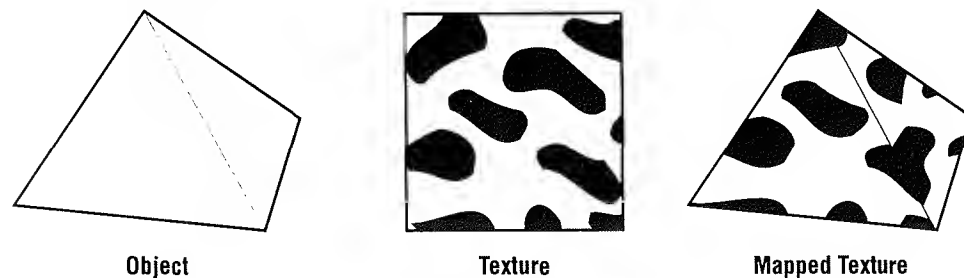


Figure 3-2 The texture-mapping process

Using Filtering in Texture-Mapping Operations

Depending on several factors—including the size of the texture being used, the 3D object's distortion, and the size of the screen image—some texels in a texture might be mapped to more than one pixel in the object's image, and some parts of the object might be covered by multiple texels.

Because a texture is made up of discrete texels, the M2 Triangle Engine performs various kinds of filtering operations to map texels to pixels in images of 3D objects. For example, if many texels correspond to a single fragment of an object, they are averaged down to fit the number of pixels into which they are mapped. If texel boundaries fall across fragment boundaries, things get even more complicated, and a weight average of the affected pixels is performed.

Replicating Textures

M2 allows you to repeat a texture over the surface of an object. Replicating a texture in this way is called *tiling*. If a texture is to be tiled, it must have a size that is a power of two in both directions. If this condition is met, a simple mask is used on the texture's final *U*, *V* coordinate (refer to Figure 3-1) before the address calculation is performed to mask out some most significant bits (MSBs).

Mipmapping

A textured object, like any other object, can be viewed at different distances from the camera. If a textured object moves away from the viewer, or if the viewer moves away from the object, the number of pixels covered by a single texel decreases. To make this change in texture mapping correlate properly with the reduction of the object, M2 filters the texture map down to an appropriate size as the object grows smaller, without introducing any visually disturbing artifacts.

To prevent the generation of such artifacts, M2 uses prefiltered texture maps called *mipmaps* (see Figure 3-3).

Mipmaps (MIP stands for the Latin *multum in parvo*, meaning “many things in a small place”) are maps that provide multiple sizes of the same texture in powers of 2. The size of a mipmap can range from 1 x 1 texel to the size of your highest-resolution map. For example, a set of four mipmaps could have sizes of 64 x 16, 16 x 4, 8 x 2, and 4 x 1.




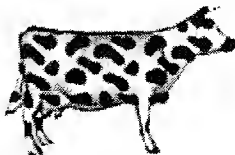




Mipmap Level	Resolution	Actual Size	Magnified
0	Full		 <i>Not Magnified</i>
1	1/2		 <i>Magnified by 2</i>
2	1/4		 <i>Magnified by 4</i>
3	1/8		 <i>Magnified by 8</i>

Figure 3-3 Mipmaps

Figure 3-3 shows how mipmaps provide multiple sizes of the same texture image. Notice that each mipmap is one-fourth the size of the next higher-level mipmap: one-half its height, and one-half its width.

Levels of Detail (LODs)

Each version of a texture’s image is called is called a *level of detail* (LOD). The original version of the texture, which has the finest detail, is called LOD 0. Other LODs are numbered upward consecutively, as shown in Figure 3-3.

M2 hardware supports the use of up to four levels of detail (LODs) at a time, but you can control which LODs are used by specifying the index of the finest LOD and the number of LODs to use.

Using Mipmaps

When you have created a set of mipmaps, here's how M2 uses them: Starting with the highest-resolution texture, the desired number of filtered and minified versions is created. Each version is reduced from the size of the previous version by a factor of two in both the U and V directions (U and V are the texture-coordinate equivalents of x and y in the world coordinate system).

When an object is texture-mapped and a set of mipmaps is available, M2 can automatically determine which mipmaps to use, on a per-pixel basis, based on the size (in pixels) of the object being mapped. When the image of the object gets small enough, M2 can switch to the next smallest mipmap to map the object's texture. When the object grows large enough, M2 can switch to the next-largest mipmap.

Mipmap Filtering

Mipmapping improves the appearance of rendered images considerably. But when you map a texel to a surface, there is rarely an exact match between the (U , V) coordinates of the texel being mapped and the (x , y) coordinates of the mipmap being used in the mapping operation. In other words, the precise texture-to-map ratios shown in Figure 3-3 are ideal ratios that actually seldom occur.

When that is the case—as it usually is—an operation known as *mipmap filtering* is often required to avoid aliasing.

Although some mipmap filter is needed in most texture-mapping operations, the amount of filtering you should perform in any particular situation can vary, depending on your needs. Also, M2 supports four different filtering modes, each designed for use in a specific kind of situation.

The four filtering modes offered by M2 are:

- ◆ *Nearest (point) filtering*, in which each texel in a texture is mapped to a surface using the best mipmap that is available, ignoring the possibility that the match may not be precise.
- ◆ *Linear filtering*, which uses two mipmaps—one that is actually too large and another that is actually too small—to average the color of each texel being matched
- ◆ *Bi-linear filtering*, which uses a similar technique to the one used in linear filtering, but applies a weighted average of the colors of the texels surrounding the texel being mapped in order to obtain a color value for that pixel that is more precise
- ◆ *Quasi tri-linear filtering*, which uses a slower but more precise algorithm to obtain an even more precise weighted value for the texel being mapped.

Each of these filtering modes has advantages and disadvantages. As you move from the fastest filtering mode (nearest [point] filtering) to the slowest, the quality of the rendered image improves, but the time required to filter the data increases. Nearest (point) filtering is the fastest mode, but offers the lowest resolution. Quasi tri-linear filtering is the slowest method, but offers the highest resolution.

Table 3-1 lists the four texture-filtering modes used in M2 and compares their speed and characteristics. Note that the bi-linear and quasi tri-linear filtering modes require a 2-by-2 array of texels each.

Table 3-1 Texture Filtering Modes

Type	Speed	Texels Required	Operation
Point	2 pix/tick	$T[U, V, n]$	None Out = $T[U, V, n]$
Linear	1 pix/tick	$T[U, V, n]$, $T[U/2, V/2, n+1]$	When a sample lies between two mipmaps, the distance of the sample from each map is used to compute a linear blend between the two maps.
Bi-Linear	0.5 pix/tick	$T[U, V, n]$, $T[U+1, V, n]$, $T[U, V+1, n]$, $T[U+1, V+1, n]$	The fractional placement within a map is used to blend four adjacent texels from one map together.
Quasi Tri-Linear	0.33 pix/tick	$T[U, V, n]$, $T[U+1, V, n]$, $T[U, V+1, n]$, $T[U+1, V+1, n]$, $T[U/2, V/2, n+1]$	A mixture between the two above modes. A bi-linear blend is performed at LOD_n , then a linear blend is performed with a texel from LOD_{n+1} .

Note: For texels at the edge of a texture, all the information required may not be present. You may be able to ignore this anomaly, or you may want to add a one-pixel guard region around the whole texture. By adding a guard region, you can guarantee that all required information is present. (A guard region is actually required for texture tiling; for details, see "Runtime Carving" on page 38.) In the M2 hardware, the address clamps in both the U and V directions, so if you do not add a guard region, replicating the last texel is the default mode.

How M2 Mipmap Filtering Works

Figure 3-4 shows the various kinds of filtering operations that are used for mipmapping in M2. The diagram is a stylized representation of a perspective view that extends from the camera out into the distance. Such a triangle could exist, say, as part of the floor of a large building.

The problem that such a triangle generates for the M2 hardware is that at some points the camera is not between two LODs. At those points, linear interpolation between maps is not possible. (It is assumed here for sake of argument that Region 3 in the diagram is the desired operating region. At that point, the camera lies between maps of different LODs, so you can perform linear or tri-linear interpolation).

If the sampling process takes you into Region 2, that means you are trying to magnify beyond the finest level of detail, which is not possible. That is the point where interpolation no longer works, and where point sampling the texture would lead to blockiness. At that point, as in the previous illustration, the best you can do is to switch to bi-linear interpolation in such a way that the fraction bits of the (U, V) coordinates are used to position a box filter over four neighboring texels.

Note that this alternative does not create any more information; it simply blurs the information. But that is preferable to the blockiness that would result from point sampling.

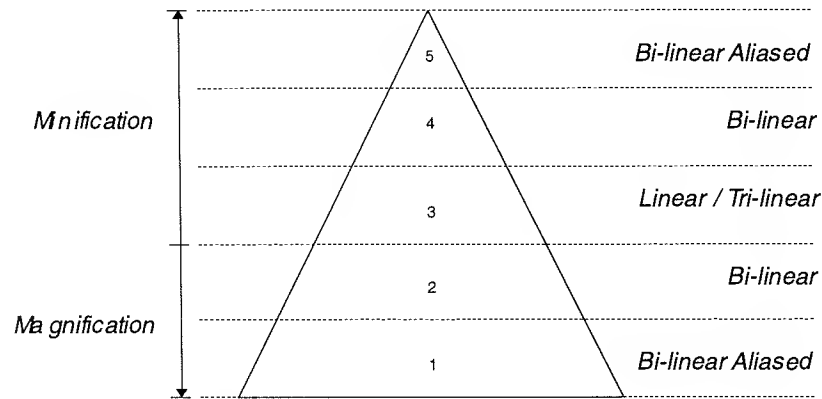


Figure 3-4 Tradeoffs in filtering operations

Monotonic Region Variations

In Figure 3-4, notice that within any one span, the region variation is *monotonic*—that is to say, once you are out of one region, no more pixels within the horizontal span you have entered can fall within that region again. It is possible, however, to jump past several regions at once. For example, you can go from Region 1 to Region 5 in one pixel step. However, you can't go from Region 3 to Region 4 and back to Region 3 again within a span.

Figure 3-4 also illustrates some problems that can arise in using mipmap filtering. Assume that the scene shown in the diagram has such a long perspective that the texture you are applying to the nearest rectangle is magnified beyond the level of detail provided by the finest mipmap available. At the same time, assume that the texture you are applying to the farthest rectangle is minified beyond the level of detail provided by the coarsest mipmap available.

Problems in Minification

For minification, a problem arises if the programmer has not provided all the LODs that are needed to bring resolution down to the 1-by-1 map. Again, the M2 hardware can detect the fact that it has fallen out of the range in which linear interpolation is possible. When that happens, the hardware falls into Region 4, where it must now either point-sample the coarsest available LOD map (an operation results in aliasing) or (once again) use bi-linear interpolation.

Because the hardware can easily detect any of the conditions that have just been described, it is possible for the hardware to select the best possible algorithm for each region. But blindly trusting the hardware to make such a selection has drawbacks that arise from the performance differences between the various interpolation schemes.

If you don't want to trust everything to the hardware, you can specify (by setting a register) exactly which algorithm to select for each region. Table 3-2 shows all the legal selections you can make. Note that regions 1 and 5 are really part of 2 and 4 respectively.

Table 3-2 Registers for specifying LOD Regions

Region	Point	Linear	Bi-Linear	Q Tri-Linear
1	•		*	
2	•		*	
3	•	•	*	•
4	•		*	
5	•		*	

Warning: For a filtering operation that involves perspectives such as those shown in Figure 3-4 on page 23, you will probably make the boundaries between regions visible if you change filter modes. When you are working with a static image, this side effect may not be very noticeable, but in a moving image the effect can be serious more serious because each region will have different motion artifacts.

Using Mipmap Filtering

Now that we have seen how mipmap filtering works, we are ready to examine each of the varieties of filtering available in M2: nearest (point) filtering, linear filtering, bi-linear filtering, and quasi tri-linear filtering. This section describes each of these kinds of filtering in more detail.

Nearest (Point) Filtering

Nearest (point) filtering is the simplest kind of filtering operation. You can always perform nearest (point) filtering.

In a nearest (point) filtering operation, the filtered texture values map directly to the color and alpha values of the texel containing the point. This point is represented by the coordinates (U, V) in the equation below.

$$LOD = [U, V, n]$$

U and V are the texel coordinates and n is the level of detail (LOD) of the mipmap being used for the mapping operation.

Linear Filtering

When a filtering operation is performed, the (U, V) coordinates for adjacent destination pixels may be such that the computed level of detail falls between two LODs. For example, assume that i is the distance of the incrementation between two horizontal texels being mapped to a surface. If $(U(i+1) - U(i))$ is 3, the computed level of detail is in between 1 and 2. In such a case, linear filtering can be used to blend the texel data between the two bounding LODs. Linear filtering is illustrated below.

$$T = T[U, V, n] \bullet (1 - \text{blend}) + T[U, V, n + 1] \bullet \text{blend}$$

n is the level of detail (LOD) of the mipmap being used for the mapping operation, just as it was in the previous equation.

Linear filtering works by taking samples of the texture at (U, V, n) and at $(U, V, n + 1)$. The sample values are combined in the ratio provided by the *blend* variable, which is computed based on the distance of the computed LOD from the finer LOD, in accordance with the preceding equation.

Requirements for Linear Filtering

Because linear filtering interpolates between two levels of detail, it cannot always be used for all pixels in a rendered triangle. In order to use linear filtering, the following requirements must be met:

- ◆ The texture being applied must have more than one LOD.
- ◆ The pixels being rendered must scale the texture such that it is larger than the finest LOD, and smaller than the coarsest LOD.

An Example of Linear Filtering

As an example of linear filtering, assume that a triangle is being rendered using a texture map that contains two levels of detail. Also assume that the dimensions of LOD 0 are 20×20 texels. That means that the dimensions of LOD 1 are 10×10 , because each subsequent LOD must be half the width and half the height of the previous LOD. So the triangle to be rendered always uses texture coordinates that range from $(0,0)$ to $(19,19)$.

Note: For the sake of simplicity, this example ignores the effects of the perspective correction factor, W , and assumes that an increase of 1.0 in triangle coordinate space maps directly to an increase of 1.0 in screen coordinates.

Now assume that this triangle is to be rendered in such a way that its screen dimensions will be larger than 20 pixels wide or 20 pixels high. If that is the case, the largest texture LOD will have to be scaled *up* to fit the triangle. In M2 terminology, we say that the *magnification filter* is used.

In contrast, if the triangle is rendered in such a way that its screen dimensions are smaller than 10 pixels wide or 10 pixels high, the smallest LOD must be scaled *down* to fit the triangle. In M2 terminology, we say that the *minification filter* is used.

Now let's consider a third case. If the triangle we are discussing is rendered in such a way that its screen dimensions are between 10 and 20 pixels in the horizontal and vertical directions, the M2 *interfilter* is used. Note that when the Triangle Engine determines which LOD is needed, horizontal LOD and vertical LOD are computed independently, so it is possible that a pixel gets magnified horizontally, and minified vertically.

There is only one case in which linear filtering can occur (or, to put it into M2 terminology, there only one case in which the *linear filter* is used). That case arises when there is a texture map larger than the rendered triangle's needs, or when there is a texture map smaller than the rendered triangle's needs. When either of those texture maps is needed, the M2 linear filter can interpolate between the two. When a texture map needs to be magnified and there is no larger texture map available to interpolate the finest LOD, or when a texture map needs to be minified and there is no smaller texture with which to interpolate the coarsest

LOD, the linear filter can be used. Because certain filter types only make sense when a texel is between LODs (in other words, when the interfilter is used), each of the three filter regions can independently select one of the available filter modes.

Note: *Linear filtering takes twice as long as nearest (point) filtering.*

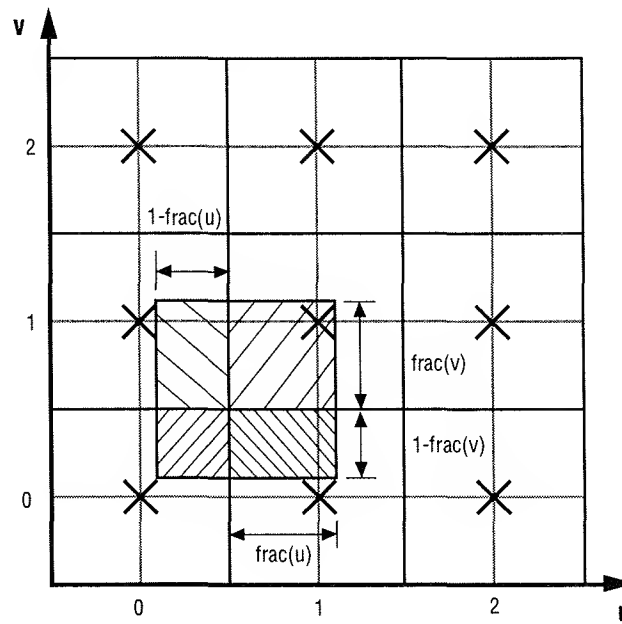


Figure 3-5 *Bi-linear filtering.*

Bi-linear Filtering

Bi-linear filtering (Figure 3-5), like nearest (point) filtering, is a type of filtering that can always be performed. Bi-linear filtering is equivalent to a one-pixel area filter that takes the weighted average from four adjacent texels, as shown below.

$$T_{out} = T[U, V, n] * (1 - fu) * (1 - fv) + T[U, V + 1, n] * (1 - fu) * fv + \\ T[U + 1, V, n] * fu * (1 - fv) + T[U + 1, V + 1, n] * fu * fv$$

In a bi-linear filtering operation, the fractional parts of (U, V) are used to position the center of the filter, as shown in Figure 3-5. Note that the texel center is at $(.0, .0)$. Given the fractional parts of U and V , the four shaded areas can be constructed. The output value is then the sum of all four fractional areas multiplied by the corresponding pixel values.

Note: *Bi-linear filtering is four times slower than nearest (point) filtering. For a precise comparison of the speeds of different filtering modes, see Table 3-1.*

Quasi Tri-linear Filtering

Quasi tri-linear filtering is a cross between linear filtering and bi-linear filtering. In a quasi tri-linear filtering operation, a bi-linear blend is performed at the closest level of detail (determined by looking at the blend value) and point sample a texel

at the further LOD. A linear blend is then performed on the two results. This procedure results in the highest-quality resampling of the texture, but is somewhat slower than a straight bi-linear blend (see Table 3-1).

Quasi tri-linear filtering provides the best possible texturing quality, but it cannot always be used for all pixels in a rendered triangle. Because quasi tri-linear filtering performs a linear blend between two LODs, it can only be specified as the interfilter.

Perspective Correction

The M2 Triangle Engine has two modes of operation. The default mode is called *Perspective On Mode*. In this mode, M2 calculates the texture locations with perspective correction. The second mode is called *Perspective Off Mode*.

Perspective On mode provides perspective-correct U and V coordinates that are appropriate for mapping a texture onto a 3D surface. In Perspective Off mode, the texture values that are passed to the Texture Mapper are the raw U/w and V/w values, before division by $1/w$. Perspective Off mode can be useful in a 2D environment when you want to use a texture as a sprite and map it directly to the screen.

See Appendix A for the register setup used to turn perspective correction on and off.

Computing the U and V Coordinates for Texture Mapping

In Perspective On mode, the U and V coordinates are typically computed as

$$(U/w) / (1/w)$$

and as

$$(V/w) / (1/w)$$

This computation results in perspective-correct U and V coordinates that are appropriate for mapping a texture onto a 3D surface. When the Perspective Off mode is selected, the $1/w$ value is not used, and the U/w and V/w coordinates are passed directly into U and V .

Using Perspective Off Mode in 2D Operations

Perspective Off mode can be useful in 2D rendering when the user wants to step through a flat texture and map it directly to the screen. It can also be useful in 3D mode at extremely large distances.

You can use Perspective Off mode when you want to generate successive integer U and V values for each succeeding pixel of a span.

The easiest way to accomplish this goal is to use Perspective Off mode, but to set the U/w and V/w values at the vertices of the triangles in such a way that the d/dx values of these parameters are the same value: namely, 1.

You can accomplish this same effect in Perspective On mode by setting the $1/w$ value as close to 1 as possible, making $(U/w) / (1/w)$ is basically equal to U/w . However, this technique precludes the use of the $1/w$ value for z-buffering in this situation. If z-buffering is not needed, using Perspective Off mode can eliminate the need for passing any $1/w$ information into the Triangle Engine at all.

How Textures Are Stored in Memory

The M2 hardware supports texture sizes of up to 1,024 by 1,024 texels and up to four levels of detail (LOD). All the LODs that you use simultaneously must fit into a 16k block of dedicated RAM within the Triangle Engine called texture RAM, or TRAM.

In M2, the TRAM must be demand-loaded by software. Each successively coarser LOD must be exactly half the size in *U* and *V* from the previous level.

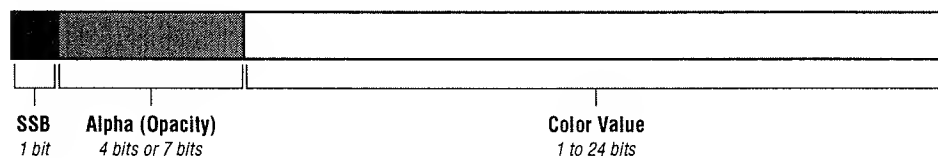
The programmer specifies the size of the coarsest LOD and the number of LODs to load. The size for the finer LODs are calculated automatically. Any aspect ratio texture may be specified.

When you load a texture into TRAM, you must load all adjacent LODs. For example, you cannot load just LOD0 and LOD2—you must also load LOD1. It is important to comply with this requirement; if you do not, the hardware produces unpredictable results.

When a texture is loaded, the *Umax* and *Vmax* register fields (see TXTUVMAX - Texture Loader Width Register (0x0004_642C)) are loaded with the dimensions of the coarsest map that is loaded. The finer mipmap sizes are given by multiplying by $2^{(LOD_{max} - LOD_N)}$ where *LODN* is the current level of detail. Each texel in a texture can contain from one to three components. If a texel has three components, they are:

- ◆ A *source selection bit* (SSB) that is generally used to select between two sets of constant registers (for more information, see "Texture Mapping"). An SSB, when used, is always 1 bit long.
- ◆ An *alpha component* that specifies the texture's opacity. An alpha component can be either 4 bit long or 7 bits long.
- ◆ A *color component* that has two possible uses. It can specify the colors used in the texture or can be treated as an index into a palette index table (see "PIP Tables"). A color component can be 1 to 24 bits long.

The length of a texel's alpha component and color component depend on two things: whether the texel is compressed or uncompressed, and what kind of information the color value of the texel contains. illustrates the three parts of a texel.



The three possible components of a texel

Texture Formats

Textures in memory may be of two types: compressed or uncompressed. Texels within a texture may be of two types: literal or indexed. The term *texture formats* encompasses all four of these texture types. Table 3-3 lists and describes all four texture formats used in M2.

Table 3-3 *Texel Types*

Type	Description
Literal	Texels are stored as real color values. Only two formats are supported: five bits per color component or eight bits per color component.
Indexed	Texels may be stored using an arbitrary number of bits per texel up to eight bits. These get expanded to 8 bits per color component by using the PIP.
Compressed	Texels are stored using a run length encoding. Multiple bits per texel (types) may be used within the texture. These are expanded to the largest texel size used when the texture is read into the internal memory. Literal formats can not be used in compressed textures.
Uncompressed	Texels are stored in one format only. The number of texels is exactly the height * width.

In addition to color information, each texel format shown in Table 3-3 may also have two more pieces of information associated with it: alpha (which may be four or seven bits), and a Source Select Bit (SSB), which is a one-bit value that has multiple uses (in general, the SSB is used to select between two sets of constant registers for each stage defined for each register; for more details, see "How PIP Tables Work.")

Table 3-4 lists all the combinations of color, alpha and SSB values that a texel can have.

Table 3-4 *Color, Alpha, and SSB Values that Can Be Assigned to a Texel*

Color	0	1	1	2	0	3	4	6	5	1	2	0	3	4	7	8	4	7	8	8	15	24	24
Alpha	-	-	-	-	4	-	-	-	-	4	4	7	4	4	-	-	7	4	4	7	-	-	7
SSB	1	-	1	-	-	1	-	-	1	1	-	1	1	-	1	-	1	1	-	1	1	-	1
Total	1	1	2	2	4	4	4	6	6	6	6	8	8	8	8	8	12	12	12	16	16	24	32

How Texels Are Stored in Memory

Uncompressed texels and compressed texels are stored in memory in different ways. Uncompressed texels are stored as arrays of texel data, but compressed texels are stored as texels of different types that are run length encoded. This section describes both these methods of storing texels in memory.

How Uncompressed Texels are Stored

Uncompressed textures are stored in memory as arrays of texel data. The number of bits per texel may vary from 1 to 32. Figure 3-6 shows how uncompressed texels are stored in memory.

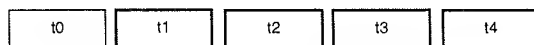


Figure 3-6 Storage format of an uncompressed texel

How Compressed Texels are Stored

Compressed textures consist of texels of different types (or formats) of texels that are run-length-encoded. This storage format allows the source texture to contain texels of different depths, so that portions of a texture that need more detail can use higher texel depths.

When texels are stored in memory, each sequence of texels that have the same type and value is called a *run*. Each new run is preceded by a control byte that specifies the run length and the type. Portions can be encoded to use less memory by switching to a different texel format on a per-run basis.

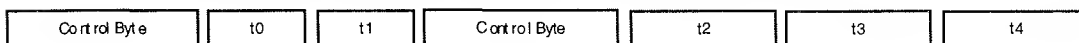


Figure 3-7 Storage format of a compressed texel

Figure 3-7 shows two run lengths of different texel types. As you can see, each run length is preceded by a control byte (in this example, the different run lengths have different depths as well).

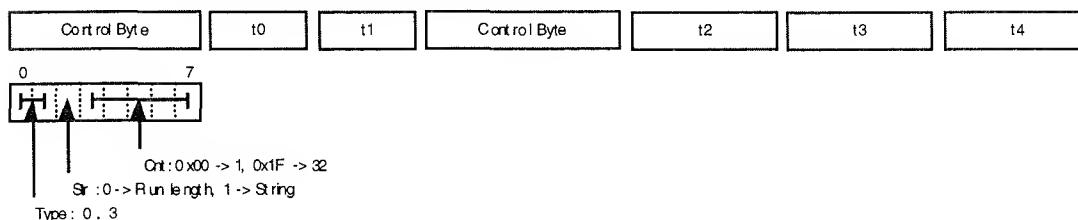


Figure 3-8 Bits in the control byte

Figure 3-8 shows the bits in the control byte that precedes each run length in a compressed texel. As the diagram shows, a control byte has three fields. These three fields are defined as follows:

- ◆ *Type (bits 0 and 1)*—This field specifies the texel type (format) of the run of texels that immediately follow the control byte. The TYPE field is simply used to select which of the *TxtLdSrcForm* registers to choose. Four types are possible. Each type is specified by the values of the four *TxtLdSrcForm* registers. The next control byte may specify a different type for its run. All types are expanded out to the format specified within the *TxtExpForm* registers.
- ◆ *Str (bit 2)*—This bit specifies two kinds of storage: *string* storage when set, and *run-length* storage when cleared. When string storage is used, the CNT field (next item in this list) specifies the number of texels that follow the control

byte. When a run is copied into TRAM, each of these texels is copied to the TRAM—that is, n texels in the source are mapped to n texels in the TRAM). When run-length storage is used, the control byte is followed by only one texel. This texel is copied into the next n locations within the TRAM, where n is encoded in the CNT field (i.e. one texel in the source gets expanded to n texels in the TRAM).

- ◆ *Cnt (bits 3 through 7)*—This field encodes the number of texels that the current control byte will generate. If n is the total number of expanded texels, then $n = \text{CNT} + 1$. If STR is set to *string*, n also specifies the number of texels following the current control byte.

Loading a compressed texture requires two steps: Run-length decoding and texel decompression. These steps work as follows:

1. *Run-length decoding* searches for control bytes and extracts the appropriate number of texels per control byte.
2. *Texel decompression* is the process that converts the four possible source texel formats to one format that is placed in the TRAM (the TRAM may only contain one texel type).

When you store compressed texels in memory, you can use source texels that are chosen from any of the legal texel types (indexed and literal) and may be mixed to some extent within the source texture (specifically, up to four different formats can be mixed and matched per texture). The restrictions on the use is that literal and indexed texels may not be mixed within one texture.

Special Settings in the Control Byte

Two special settings can appear in the control byte shown in Figure 3-8:

- ◆ If the setting of the *Str* field is 0 and the setting of the *Cnt* field is 0, 0, the run that follows the control bit has a run length of 1. This setting is reserved, and should not be used in applications.
- ◆ If the *Type* field is programmed as transparent—that is, if *TxtLdSrcForm[TYPE].Trans = 1*—the *Str* and the *Cnt* fields are concatenated to form one 6-bit count value. No source texels follow the control byte. When a run uses a transparent format, the source texels are retrieved from a corresponding user-settable constant. See the "Texture Mapper Pipeline Register Definitions" section, later in this chapter, for more information.

If a texture in main memory is run-length encoded, it can be a mixture of the preceding formats (although literal and indexed formats cannot be mixed). The texture within the TRAM may only contain one format at a time. It is the job of the loader to ensure that only one texel type is used in the TRAM. The filtering and blending stages all use a consistent 1, 8, 8, 8, 8 format for SSB, Alpha, Red, Green and Blue.

PIP Tables

A PIP table (palette index table) is a component of the M2 Triangle Engine that is used to look up information about textures that are stored as indices. From the information stored in a PIP table, you can reconstruct the stored texels at load time.

Every PIP table has 256 entries. Each PIP-table entry is a 32-bit value constructed just like a texel. When you use a PIP table to look up information about a texture, the input texel is used as an offset into the table that contains the desired palette for the texture. The entries in a PIP table do not have to be expanded out to full 32-bit colors before filtering or texture application can be performed.

Figure 3-9 shows how a texel's color component can be used as an index into a PIP table.

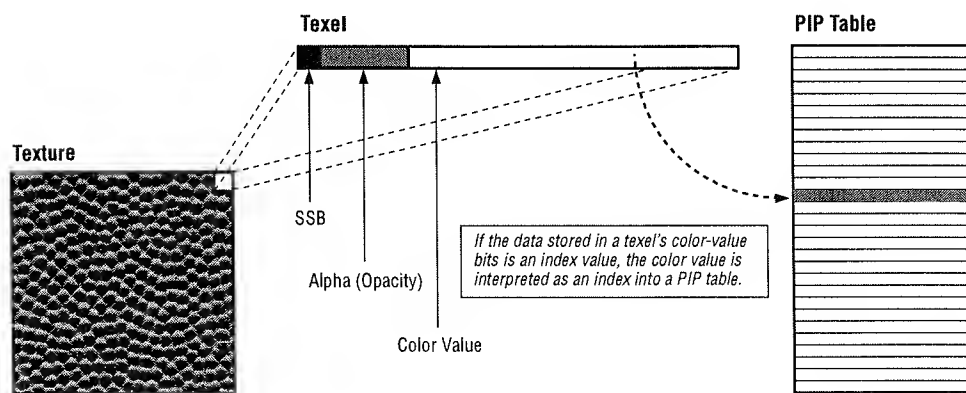


Figure 3-9 How textures and texels work with PIP tables

PIP tables have a fair amount of built-in flexibility. The format of input texels can range from 0 to 8 bits per texel. An alpha value may also be present. Adding an alpha value increases the number of available formats, but the alpha value is not used to reference into the PIP table. The way in which the alpha value is used depends on other bits in the texel; for details, see "How PIP Tables Work."

The Segment Pointer

Because texel formats that require less than 8 bits per texel do not make full use of the 32-bit entries provided in a PIP table, the design of the PIP table also provides a *segment pointer* that can select from a variable number of segments. For example, a PIP table can be used in 1-bpt (bit-per-texel) mode, in which there are 256 possible segments, or in 8-bpt mode, in which there is only one segment.

By using segments, you can load information for a large number of textures (or variations of textures) in a single PIP table. By using this strategy, you can avoid the need to keep unloading and reloading different PIP tables. To divide a PIP table into segments, you use a shift, a mask and a constant. You apply a shift applied to the input index, and then you use a mask to select between the shifted input index and a constant on a bit-by-bit basis.

What Can Be Stored in a PIP table

Seven bits of alpha information and one control bit (SSB) can also be stored in a PIP table. SSB, alpha, and color can each independently come from any one of these three sources:

- ◆ from the PIP Table (PIP RAM)
- ◆ from a constant
- ◆ from data directly within the source texel.

In order for any of these components to come directly from the source texel, the source texel's format must contain literal data for that component. In other words, if you want to take the alpha component from the source texel, the texture must be of a format that contains either 4 or 7 bits of alpha (see Table 3-4).

You cannot select the source texture as the output of a color value unless the input value for the color was a literal value.

How M2 Interprets Texel Data

If the color data in a texel is interpreted as a PIP offset, resulting in the texel's being passed on to a PIP table, the PIP table can also interpret the texel's color data in several ways, depending on how various texture-related attributes are set. The PIP unit can leave the texel's data values just as they are, or it can look up colors in a color table. It can also treat each of the three components in the texel as a constant. These choices provide great flexibility for certain kinds of effects.

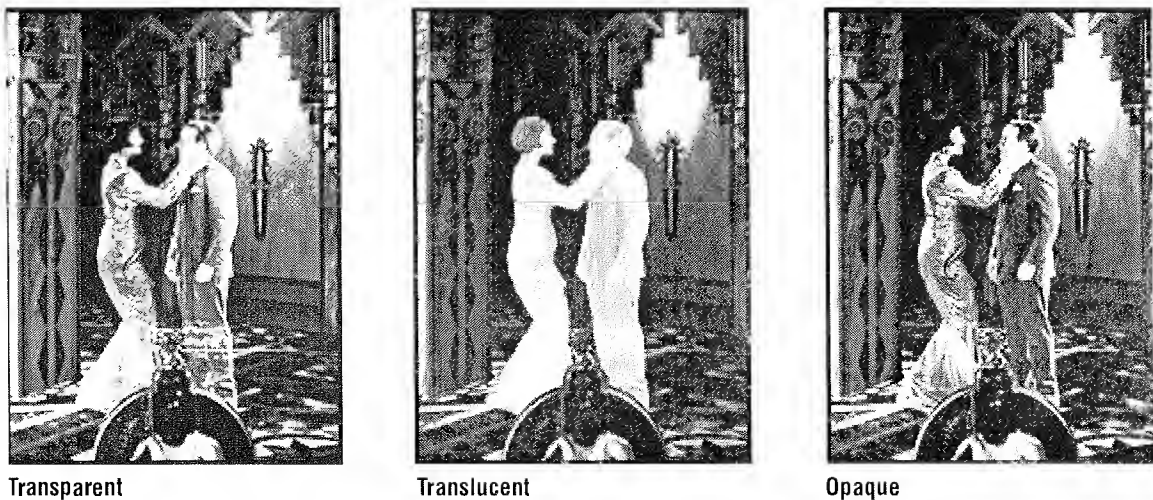


Figure 3-10 *Rendering a ghost*

How PIP Tables Work

The color, alpha, and SSB output of the PIP mapping stage can be obtained from one of three places: it can come from the texel value retrieved from TRAM, from the PIP itself, or from the two constants defined by the user. The utility of PIP mapping can be demonstrated by a simple example.

Let's look at a texture representing a ghost (see Figure 3-10). Assume that the ghost has eight colors and that you want the colors to fade in gradually over a certain number of frames. You can represent this kind of a texture can be represented using four bits per texel —three bits per texel for the color index and one bit per texel for the SSB.

In the texture-mapping example shown in Figure 3-10, the application sets color to come from the PIP table, alpha to come from the constants, and the SSB to come from the TRAM. The alpha component of the first color constant is used for texels

that have an SSB value of 0, and the alpha of the second color constant is used for texels that have an SSB value of 1. The application needs only to change the values of the color constants between different frames to fade out the ghost.

Also, the dynamic range of the alpha value can be up to seven bits. So PIP mapping can be used for certain kinds of animations while achieving significant data compression as well. The output of the PIP mapping stage is used for further processing, as described below.

The color, alpha, and SSB components of `PipConsts` are packed into a `uint32` data type, as shown in Figure 3-11.

SSB	Alpha	Red	Green	Blue
1 bit	7 bits	8 bits	8 bits	8 bits

Figure 3-11 Bit packing of a Color

See [Appendix A](#) for an example of the register setup for a PIP load.

Texture Application Blending

The final stage of the texture pipeline is the Texture Application Blender (TAB). This is the stage when the primitive color and alpha components of each texel are blended with the filtered texel value obtained from the filtering stage. Primitive color refers to the color a pixel would have been painted in the absence of texturing. Primitive color is often the result of lighting and shading computations. The blending process for color may be composed of either a LERP or a multiply :

- ◆ LERP— $D = (1 - C) * A + C * B + f(A, B, C)$
- ◆ Multiply— $D = A * B + f(A, B)$

—where A , B , and C are chosen from a list of C_t , A_t , C_{iter} , A_{iter} or constants. $f()$ is an error function. C_t is the color of the texel (from the PIP table or from a source indexed from the PIP table; see “What Can Be Stored in a PIP table” on page 32). A_t is the alpha from the texel. C_{iter} is the iterated color—that is, the interpolated color from the color data at the vertices of the triangle. A_{iter} is the iterated alpha. See [Appendix A](#) for an example of how these calculations work.

Multiplying Color and Alpha Components

When you perform LERP and multiplication operations, the input values of A , B , and C are assumed to have a range of $[0, 1]$ —that is, the legal input value includes both 0 and 1. Because A , B , and C are represented by 8-bit numbers, the hexadecimal value `0x00` represents 0 and hex `0xFF` represents 1.0.

Note that in the three application modes described under the following section “Application Modes,” these values result in an error if a simple multiplication operation is performed. For example,

$$0xFF * 0xFF = 0xFE$$

Clearly, the result of this simple multiply should be `0xFF`, not `0xFE`. To compensate for this anomaly, the error function $f()$ is added in. This operation applies an offset to the calculation.

Ideally, this correction should spread out the error term over the full range of output values. However, for ease of implementation, one of the inputs is passed directly to the output if the other input is '1' (0xFF). For the LERP, *A* is passed out directly if *C* is 0x00, and *B* is passed out directly if *C* is 0xFF. For the multiply, *A* is passed out directly if *B* is 0xFF, and *B* is passed out directly if *A* is 0xFF. The case of *A* and *B* both being 0xFF is left as an exercise for the reader.

The only possible function that is applied to the alpha channel is a multiply. Again, this operation passes out an unmodified value if one of the inputs is 1 (0x7F).

Application Modes

Both alpha information and color information can bypass the Texture Application Blender altogether, in which case the output may be either the textured output or the iterated output.

There are three typical application modes: *Modulate*, *Decal* and *Blend*. They are defined as follows:

- ◆ **Modulate**—This is used to generate effects such as a light source illuminating portions of a surface:

$$Cti = Ct * Citer; Ati = At * Aiter$$

- ◆ **Decal**—Here the alpha in the texture map is used to superimpose a texture onto a shaded surface. The texture will not have any effect of lighting. This is useful for effects such as stenciling lettering onto the side of an object:

$$Cti = (1 - At) * Citer + At * Ct; Ati = Aiter$$

- ◆ **Blend**—Here the color information in the texture is used to blend between a shaded surface and a background color. This is useful for effects such as transparency. For example, if we are trying to show the inside of a 3D object, such as a person, the bone would be represented by the constant color in the center and the organs surrounding the bone could be shaded, but with the bone still showing through (i.e. the organs are partially transparent):

$$Cti = (1 - Ct) * Citer + Ct * Cconst; Ati = At * Aiter$$

Texture Mapping Step-by-Step

To boil it all down, M2 texture mapping requires a sequence of three operations:

1. When a particular pixel is to be painted on the screen, texel data that corresponds to the pixel data being used is retrieved from the texture. This texel data value is passed through a PIP module for further mapping. The PIP table may leave the texel data values as they are, perform a color-table lookup, or use a constant for each of the color, alpha, and SSB (source select bit) components. This strategy provides great flexibility for certain kinds of animations.
2. Texture filtering determines the level of detail (LOD) for adjacent pairs of (*U*, *V*) values to reference an appropriate texture map. It performs blending of the color/alpha values from the LODs (a) between adjacent texture maps for linear sampling, or (b) within one LOD for bi-linear sampling, or (c) a mixture of the two for quasi tri-linear sampling.

3. Texture application blending applies the texture values obtained after filtering to the color and alpha values of the triangle pixel according to the blend style specified.

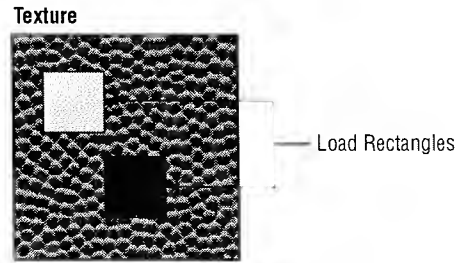


Figure 3-12 *Placing Load Rectangles inside a texture*

Loading Textures

The Texture Mapper contains a DMA engine for loading the Texture RAM (TRAM). For uncompressed textures, the maximum rate at which the TRAM can be loaded is 200MB/sec. The loader is also capable of decompressing a 3DO proprietary compressed texture format that is based around run-length encoding (RLE).

The texture loader performs four main tasks:

- ◆ *MMDMA (Memory to Memory DMA)*—This variety of load can be used to copy a certain number of bytes from main memory into the TRAM or PIP. For textures which do not require special loading operations, such as loading only a sub-rectangle of a texture in RAM into the TRAM, this load mode is the simplest to use.
- ◆ *Uncompressed Texture Load*—This kind of load is used to copy a tile from a source texture within main memory into the TRAM. The source texture must be in uncompressed format. The texel format can be any of the legal M2 formats. For uncompressed textures, a rectangular portion of a texture can be loaded.
- ◆ *Compressed Texture Load*—M2 supports run-length encoded textures. During compressed texture load, a source texture within main memory is decompressed into the TRAM.
- ◆ *PIP Load*—This type of load is used to load the contents of the PIP with a table stored in main memory.

The M2 texture loader is programmed in two stages: by first setting up the loader state and then issuing the TLD command in the TEDCntl register, or using the CLT_TxLoad() macro.

Both these operations can be performed in the command list. Consequently, texture loads can be performed without any overhead from the operating system.

The loader state consists of first selecting the required mode (see the preceding list) within the *TxtLdCntl* register. Each mode requires a different number of loader registers to be setup as well. MMDMA requires only three other registers to be programmed. Compressed texture load requires up to 13 registers to be programmed.

Loading/decompression does not work in parallel with the rest of the Pipeline (that is, the internal hardware pipeline that exists within the Triangle Engine). This is partly because it requires use of the same memory read and write buffers that are used by the destination blender, but it is mainly because the texture load process can swamp the main memory bandwidth while it is in operation. Registers are also shared between the loader and the main pipeline. For correct operation therefore, a SYNC instruction must be placed in the instruction list before any loader setup or the TLD instruction.

Loading Textures by Using MMDMA

MMDMA texture loads require the least amount of runtime setup of the M2 texture loader. In order to use this load mode, however, texture data has to appear in RAM exactly as it should appear in the TRAM. For example, if 3 texture maps with 4 LODs each must be loaded into the TRAM, and all of the texel data from these 12 maps fits within 16K of data, a single MMDMA load can be used to retrieve all 12 textures into the TRAM. To use MMDMA, specify the address in memory of the block of data to DMA in the `TxtLdSrcAddr` register, the byte count to load in the `TxtCount` register, and the word-aligned TRAM load offset in the `TxtLdDstBase` register.

Loading Uncompressed Textures

Uncompressed textures are stored as packed arrays of texels within main memory (see "How Texels Are Stored in Memory"). All the uncompressed texels stored in an array are of the same type, and the M2 hardware can automatically load a smaller tile of the source texture. Each LOD must be stored separately within main memory, and be loaded to a different base address within the TRAM.

Figure 3-13 shows how an array of uncompressed texels is stored in memory.

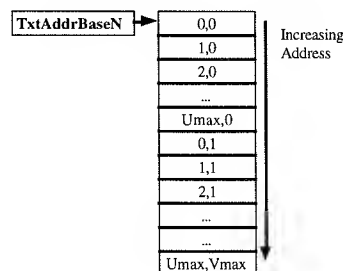


Figure 3-13 An array of uncompressed texels ready for loading

When an application loads an uncompressed texture, the application must point to the start of the first texel to be loaded. To support runtime texture carving (see "Runtime Carving"), and because the source texture can have any of the legal texel depths, the first texel can start on an arbitrary bit boundary. Consequently, the application must specify the bit-aligned address of the first source texel to be loaded (this might not be the same as the start of the texture itself).

The bit address of the first texel to be loaded is the concatenation of `TxtLdSrcBase` and the `SrcBitOff` field from the `TxtLdCntl` register:

$\{ \text{TxtLdSrcBase}[7:31], \text{SrcBitOff}[0:2] \}$

The destination address within the TRAM must be 32-bit word-aligned, and is programmed within the *TxtLdDstBase* register. The source and destination textures can have different strides. These are programmed as bit strides in the *SrcRowBits* and *DstRowBits* fields of the *TxtLdWidth* register. In addition, the total number of rows to be loaded must be programmed in *TxtRowCnt*.

When the TLD instruction is issued, the loader first accesses the byte containing the start texel. The *SrcBitOff* offset is then applied. Next, the loader copies *TxtDstBitCnt* bits into the destination buffer and does the wrapping to re-align the data.

At the end of a line, the source memory pointer is advanced to the start of the next line. This operation is performed by advancing the source pointer by $(\text{TxtSrcBitCnt} - \text{TxtDstBitCnt})$. The operation is performed *TxtRowCnt* times.

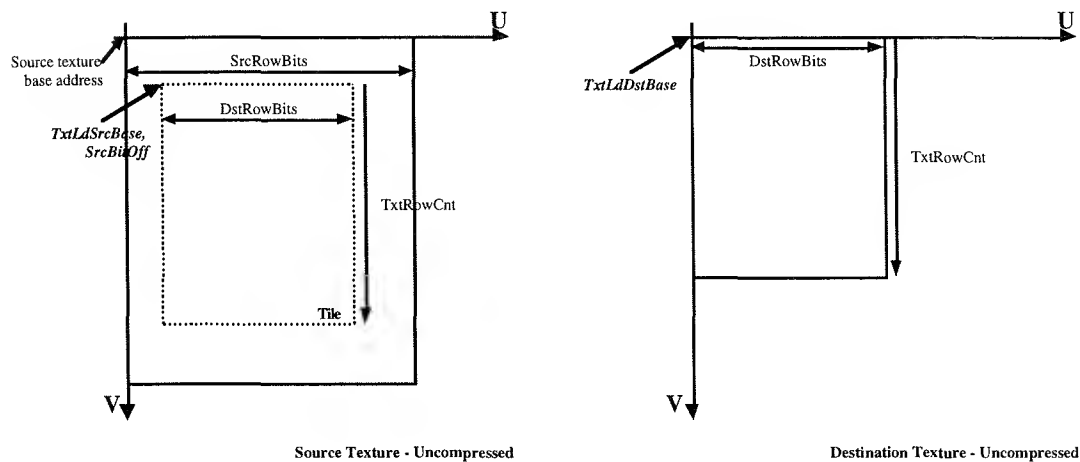


Figure 3-14 Uncompressed Loader Setup

See Appendix A for an example of how to load a texture.

Runtime Carving

When a texture is too large to fit within the internal TRAM, the texture (and associated geometry) must be broken into several smaller pieces. This disassembly is supported in hardware by the loader. The application sets the memory base address and also provides a *U* and *V* offset, along with the *V* dimension of the portion to load. This operation allows parts of textures to be loaded automatically.

Although runtime carving is simple in theory, there is one complication you should be aware of. The complication is that bi-linear filtering (as well as tri-linear filtering) require a 2-by-2 array of texels. For texels that are on the edge of a tile, there is no more information present to do the filtering. Hence the tiles must have a *guard region* that is one texel wide around all four edges to allow the joins in tiles to be seamless. When a guard region has been set up, you must ensure that none of the triangle vertices fall within the guard region. For more details about guard regions, see "Mipmap Filtering."

Compressed Texture Load

The M2 texture mapper decompresses textures by reading the control bytes within the compressed texture, and processes the information following each control byte. The processing consists of expanding out run-lengths and resolving different texel formats before placing the data into the TRAM.

The setup for compressed texture load is similar to that of uncompressed loads. The difference is that the texel formats are important during decompress. Also, runtime carving of compressed textures is not supported.

When loading compressed textures, the application provides the base of the compressed texture in `TxtLdSrcBase`, the base of the destination (uncompressed) map in `TxtLdDstBase`, the width of the source and destination in texels within `SrcRowTex` and `DstRowTex`, and the total number of rows to be loaded in `TxtRowCnt`. In addition, the texel formats must be specified as well. This is setup in the four `TxtLdSrcForm` registers and the `TxtExpForm` register. The `TxtLdSrcForm` registers specify the format of the four possible texel types within the source texture in main memory. The `TxtExpForm` register specifies the format of the texels that are to be placed into the TRAM.

See Appendix A for an example showing which registers must be programmed for a compressed texture load.

Texel Expansion

Because multiple texel formats can be present within the source texture, the formats must be expanded out to a common format before you can place them in the output buffer. The desired TRAM texel type is programmed within `TxtExpForm`. All source formats are expanded out to this type. The expansion works by first determining if each channel that is present in the output (SSB, alpha, color) is present in the source. If a certain channel is not present in the source (but is required in the output), then that data is taken from `TxtLdConst[TYPE]`, where `TYPE` is the two-bit index in the control byte. If the data is present in the source, but of a different depth, then the source channel is expanded as described below. The SSB, alpha and color information are handled independently, as follows:

- ◆ *SSB*—If the source texel has an SSB, that SSB is passed through to the expanded texel. If the source texel doesn't have an SSB, but the expanded texel does, a constant SSB is used (`TxtLdConst[TYPE].SSB`)
- ◆ *Alpha*—The source texel can contain 0, 4, or 7 bits of alpha information. If the expanded texel format has more than seven bits of alpha information, the source alpha must be expanded. If the source contains no alpha information, you can expand the source alpha by using a constant alpha (`TxtLdConst[TYPE].alpha`). If the source does contain alpha information, you can expand the source alpha by replicating the MSBs into the LSBs. In the case of constant alpha, either 7 bits or 4 bits are taken from the constant register, depending on the expanded format. Note that for 4-bit output, the top 4 bits of the constant alpha are chosen. Replication can be used only to go from 4 bits to 7 bits of alpha—the top three MSBs of the 4-bit source alpha are replicated into the bottom three LSBs of the expanded alpha.
- ◆ *Literal color*—A literal color can be either 555 or 888. Expansion from 555 to 888 is done by replication of MSBs. If no color is present in the source, then `TxtLdConst[TYPE].red`, `TxtLdConst[TYPE].green`, and `TxtLdConst[TYPE].blue`

are chosen. The top five bits of the constant are chosen for 5-bit output, in much the same way that output is chosen for alpha.

- ◆ *Indexed color*—Source and expanded index depths can be [0, 1, 2, 3, 4, 5, 6, 7, 8] bits. All of these can be expanded to any larger size (up to 8 bits, of course, which is the size of the PIP). The expansion is done by adding an offset (specified in *TxtLdConst[TYPE].index*) and masking to the appropriate depth. If no index is present in the source, then *TxtLdConst[TYPE].index* is used directly.

A special case occurs if the control-bit type is specified as transparent (see "Special Settings in the Control Byte"). In this case, no information follows the control byte. The appropriate color information is provided by one of the constant registers (*TxtLdConst[TYPE]*). Transparency is usually indicated by either zero SSB or alpha. The destination blender is programmed to use either of these values as a transparency flag.

See Appendix A for the register setup used when doing a transparency.

Texture Mapper Pipeline Register Definitions

This section describes the individual registers used to control the texture unit. The section provides descriptions of each of the registers, the address of the register, the CLT macro to set the register and the register layout.

The first half of this section describes the texture mapping registers, which control how a pre-loaded texture is to be interpreted and applied to geometry. The second half describes the registers used to load a texture.

Command Registers, Fields, and Macros

Each command register can be broken down into fields. For example, for a register named *XXX*, the macro *CLA_XXX* provides the data word to be written to the register with the arguments converted and packed into the register fields. The macro *CLT_XXX* takes a pointer to a pointer as the first argument. It writes the argument word to the command list and advances the pointer.

Table 3-5 Register Memory Map

Address	Access	Name	Information
0x000C_0000-0x000C_3FFF	RW	TRAM	Texture RAM
0x0004_6000-0x0004_63FF	RW	PIP	PIP RAM
0x0004_6404	RWSC	TxtLdCntl	Texture Mapper Load Control Register
0x0004_6408	RWSC	TxtAddrCntl	Address Generation Control Register / Filter modes
0x0004_640C	RWSC	TxtPIPCntl	PIP Control Register
0x0004_6410	RWSC	TxtTABCntl	Texture Application Control Register
0x0004_6414	RW	TxtLODBase0	LOD 0 Base Address
0x0004_6418	RW	TxtLODBase1	LOD 1 Base Address
0x0004_641C	RW	TxtLODBase2	LOD 2 Base Address
0x0004_6420	RW	TxtLODBase3	LOD 3 Base Address
0x0004_6424	RW	TxtLdSrcBase TxtMMSrcBase	Texture decompressor source base address MMDMA Source Base
0x0004_6428	RW	TxtCount TxtLdRowCnt TxtLdTexCnt	Byte Count for MMDMA and PIP load Row Count for uncompressed texture load Texel Count for compressed texture load
0x0004_642C	RW	TxtUVMax TxtLdWidth	Texture Size Register Texture Loader Width Register
0x0004_6430	RW	TxtUVMask	Texture Mask Register
0x0004_6434	RWSC	TxtSrcType01	Source Description Registers - Type 0 and 1
0x0004_6438	RWSC	TxtSrcType23	Source Description Registers - Type 2 and 3
0x0004_643C	RWSC	TxtExpType	Expanded Type Description Register
0x0004_6440	RW	TxtConst0 TxtPIPCnst0	Source Expansion Constant Register - Type 0 PIP Constant color - SSB = 0

Address	Access	Name	Information
0x0004_6444	RW	TxtConst1 TxtPIPConst1	Source Expansion Constant Register - Type 1 PIP Constant color - SSB = 1
0x0004_6448	RW	TxtConst2 TxtTABConst0	Source Expansion Constant Register - Type 2 Texture Application Constant color - SSB = 0
0x0004_644C	RW	TxtConst3 TxtTABConst1	Source Expansion Constant Register - Type 3 Texture Application Constant color - SSB = 1

Warning: Some registers are used for different purposes during the rendering and loading stages.

Texture Generation Registers

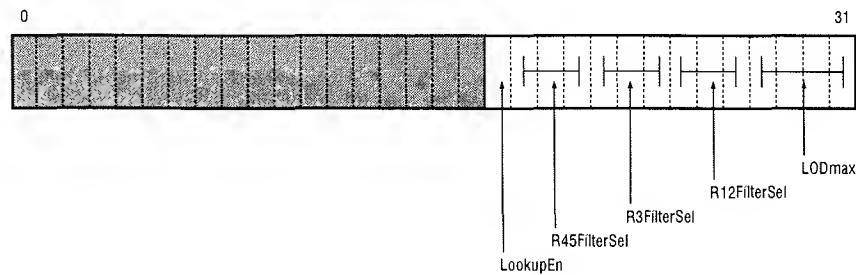
The following paragraphs list and describe the texture units' texture generation registers.

TXTADDRCTL - Address Control Register (0x0004_6408)

The address control register specifies the way in which textures are accessed.

CLT_TXTADDRCTL (pp, textureenable, minfilter, interfilter, magfilter, lodmax)

Field	Description
TEXTUREENABLE	Enables the texture lookup process
MINFILTER	Minification filter modes
POINT BILINEAR	
INTERFILTER	Inter filter modes
POINT LINEAR1 BILINEAR TRILINEAR	
MAGFILTER	Magnification filter modes
POINT BILINEAR	
LODMAX	Number of LODs present -1



The bits shown in the preceding figure are defined as follows:

- ◆ *LookupEn*—This bit enables the texture lookup process. If the *LookupEn* bit is disabled, the texture mapper does not generate any stalls to the span walker, irrespective of the texel depth and filter mode. This bit is essentially a user texture enable bit, with one exception: The texture blender is not set up to pass iterated color through by default.
- ◆ *LODmax*—Specifies the number of LODs present.

LODmax[0:3]	Definition
0x0	1 LOD
0x1	2 LODs
0x2	3 LODs
0x3	4 LODs
0x4 - 0xF	reserved

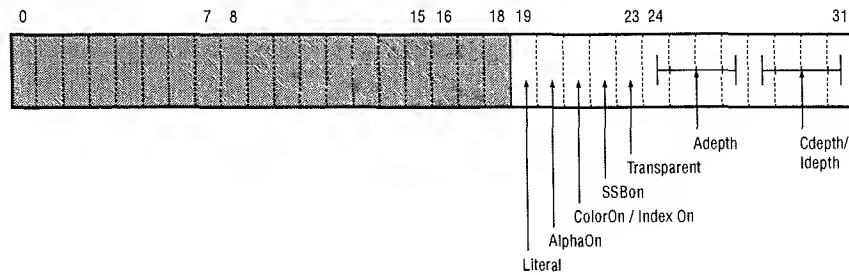
- ◆ *FilterSel*—Specifies the filter modes for the various regions. Each region may have a different filter mode selected.

FilterSel[0:2]	Definition
0x0	Point
0x1	Linear
0x2	Bilinear
0x3	Quasi-Trilinear
0x4 - 0x7	reserved

TXTEXPTYPE - Expansion Type Register (0x0004_643C)

This register describes the texel format within the texture RAM. It is used by the address generation logic to determine the depth of the texels, and by the lookup logic to unpack the data from the TRAM into SSB, color and alpha channels.

```
CLT_TXTEXPTYPE(pp, cdepth, adepth, istrans, hasssb, hascolor,
               hasalpha, isliteral)
```



The bits shown in the preceding figure are defined as follows:

- ◆ *Cdepth*—The number of bits per color component for literal formats. Only a few are valid:

Cdepth	Definition
0x0 - 0x4	reserved
0x5	5bits per color
0x6 - 0x7	reserved
0x8	8bits per color
0x9 - 0xF	reserved

- ◆ *Idepth*—The number of bits of index for indexed formats. Only the following are supported:

Idepth	Definition
0x0	reserved
0x1 - 0x8	1 - 8bits index
0x9 - 0xF	reserved

- ◆ *Adepth*—The number of bits of alpha. Only a few are valid:

Adepth	Definition
0x0 - 0x3	reserved
0x4	4bits alpha
0x5 - 0x6	reserved
0x7	7bits alpha
0x8 - 0xF	reserved

- ◆ *Transparent*—Not used during texture lookup
- ◆ *SSBon*—Specifies whether an SSB is present.

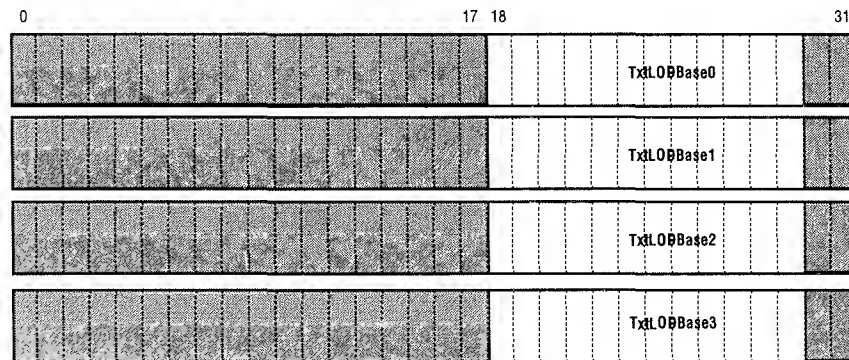
- ◆ *ColorOn, IndexOn*—Specifies whether color is present for literal texel types, or an index is present for indexed types (There are combinations that have no color bits —see below).
- ◆ *AlphaOn*—Specifies whether alpha is present.
- ◆ *Literal*—If set then the type is a literal format. If not set, then the type is indexed. See Table 3-4 for the legal texel values.

TXTL0DBASE0-3 - Texture Base Registers (0x0004_6414 - 0x0004_6420)

The fields *TxtLODBase0* through *TxtLODBase0* shown in the following table are the mipmap base address registers for texture lookup. For correct operation the mipmap addresses should be 32-bit word aligned, i.e. bits 30 and 31 should be zero. This is enforced in hardware for *TxtLODBase1-3*, but not for *TxtLODBase0*, as this register is shared with the loader.

CLT_TXTL0DBASEn (somevalue)

Field	Description
TXTL0DBASE0	Base address for LOD0
TXTL0DBASE1	Base address for LOD1
TXTL0DBASE2	Base address for LOD2
TXTL0DBASE3	Base address for LOD3

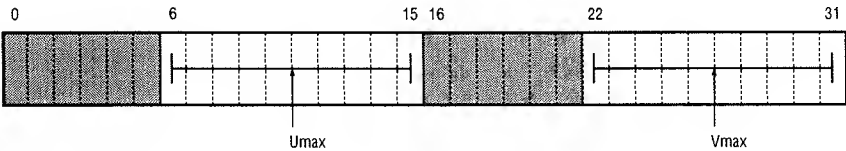


TXTUVMAX - Texture Loader Width Register (0x0004_642C)

The texture size register specifies the size of the decompressed texture during texture lookup. In this case, *UMAX* and *VMAX* (see the following table) specify the limits of the coarsest mipmap present. If *LODmax* is set to 0, this register refers to LOD0. If *LODmax* is set to 3, this register refers to LOD3. *UMAX* and *VMAX* are 10-bit quantities and should be set to the width of the texture (expressed in texels) minus 1.

CLT_TXTUVMAX (pp, umax, vmax)

Field	Description
UMAX	Width of destination buffer in bits
VMAX	Width of source buffer in bits

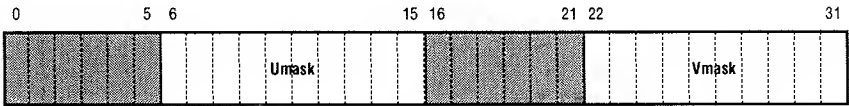


XTUVMASK - Texture Mask Register (0x0004_6430)

The texture mask is used for texture replication. Bits set to '0' in this register will be masked off during address calculation. For normal operation, this register should be programmed with 0x3FF for both masks. Note that for tiling to work properly as a result, the source texture must be a power of two in U and V. See Appendix A for the register setup used to tile a texture.

CLT_XTUVMASK (pp, umask, vmask)

Field	Description
UMASK	Enables mask
VMASK	Enables mask

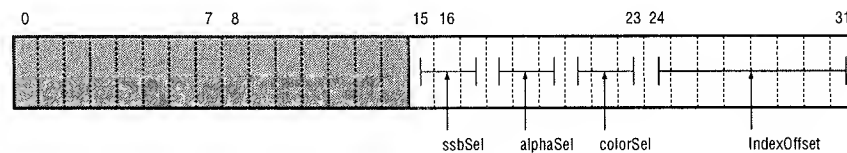


TXPICNTL - PIP Control Register (0x0004_640C)

This register controls the operation of the PIP circuitry. For the constant, the input SSB from the TRAM lookup chooses between *TxtPIPConst0* and *TxtPIPConst1* registers. Note that if *colorSel* is 0x1 (meaning that the PIP RAM is bypassed), the SSB or Alpha output should be chosen from either options 0x0 or 0x1 above for correct operation. If there is no SSB in the source texture, the Lookup section passes SSB as 0—so, in this case, *TxtPIPConst0* is always selected.

CLT_TXPICNTL (pp, pipssbselect, pipalphaselect, pipcolorselect, pipindexoffset)

Field	Description
PIPSSBSELECT CONSTANT TEXTURE PIP	Select PIP SSB module output
PIPALPHASELECT CONSTANT TEXTURE PIP	Select PIP Alpha module output
PIPCOLORSELECT CONSTANT TEXTURE PIP	Select PIP color module output
PIPINDEXOFFSET	Index offset



The bits shown in the preceding figure are defined as follows:

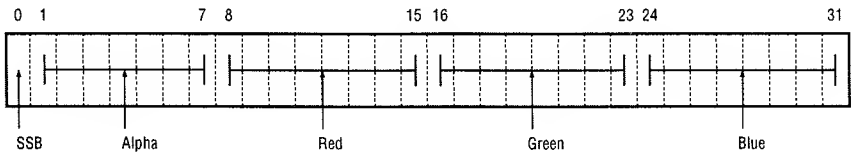
- ◆ *IndexOffset*—This field is added to the input index before accessing into the PIP RAM. This can be seen as a base pointer from where PIP accesses are to occur. The generated index wraps from 255 -> 0 on an overflow.
- ◆ *ssbSel, alphaSel, colorSel*—These fields specify the output of the PIP module for SSB, alpha and color.

ssbSel, alphaSel	Definition
0x0	Constant
0x1	Texture Cache
0x2	PIP
0x3 - 0x7	reserved

PIP Constant Registers (0x0004_6440 - 0x0004_6444)

The SSB from the TRAM Lookup is used to select between these two registers. The output for SSB, alpha or color can be independently controlled to be from the selected register. Note that if no SSB is present in the source texture, then TxtPIPConst0 is always selected.

CLT_TXTCONSTn (pp, blue, green, red, alpha, ssb) n = 0,1



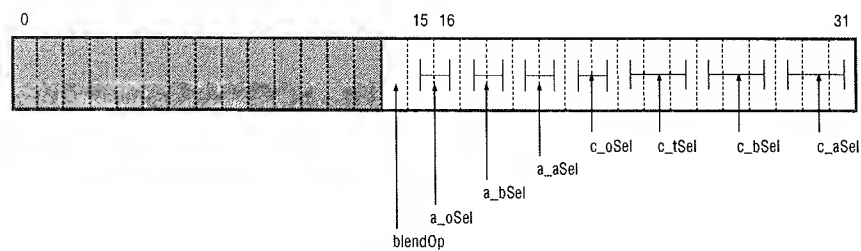
TXTTABCNTL - Texture Application Control Register (0x0004_6410)

This register controls the data path of the texture application blender. It controls the input selects to the color LERP function, and the output selects for the alpha and the color channels independently.

CLT_TXTTABCNTL (pp, firstcolor, secondcolor, thirdcolor, firstalpha, secondalpha, colorout, alphaout, blendop)

Field	Description
FIRSTCOLOR PRIMALPHA PRIMCOLOR TEXALPHA TEXCOLOR CONSTALPHA CONSTCOLOR	Select first color input
SECONDCOLOR PRIMALPHA PRIMCOLOR TEXALPHA TEXCOLOR CONSTALPHA CONSTCOLOR	Select second color input
THIRDCOLOR PRIMALPHA PRIMCOLOR TEXALPHA TEXCOLOR CONSTALPHA CONSTCOLOR	Select third color input
FIRSTALPHA PRIMALPHA TEXALPHA CONSTALPHA	Select first alpha input
SECONDALPHA PRIMALPHA TEXALPHA CONSTALPHA	Select second alpha input

Field	Description
COLOROUT PRIMCOLOR TEXCOLOR BLEND	Select color output
ALPHAOUT PRIMALPHA TEXALPHA BLEND	Select Alpha output
BLENDOP	Control the LERP function



The bits shown in the preceding figure are defined as follows:

- ◆ *c_aSel, c_bSel, c_tSel*—Control the inputs to the color LERP function.

c_aSel, c_bSel, c_tSel	Definition
0x0	Aiter
0x1	Citer
0x2	At
0x3	Ct
0x4	Aconst
0x5	Cconst
0x6 - 0x7	reserved

Note: The constant refers to the two constant registers: *TxtCATIConst0* and *TxtCATIConst01*. They are selected by the SSB coming into the blender from the filter.

- ◆ *a_aSel, a_bSel*—Control the inputs to the alpha Multiplier function.

a_aSel, a_bSel	Definition
0x0	Aiter

a_aSel, a_bSel	Definition
0x1	At
0x2	Aconst
0x3	reserved

- ◆ *c_oSel, a_oSel*—These fields control the datapath mux at the output from the blender. The two are independent.

c_oSel	Definition
0x0	Citer
0x1	Ct
0x2	Blend output
0x3	reserved

a_oSel	Definition
0x0	Aiter
0x1	At
0x2	Blend output
0x3	reserved

Note: If options 0 or 1 are selected above, then the inputs to the LERP block and alpha multiply are don't care.

- ◆ *blendOp*—Controls the function of the LERP.

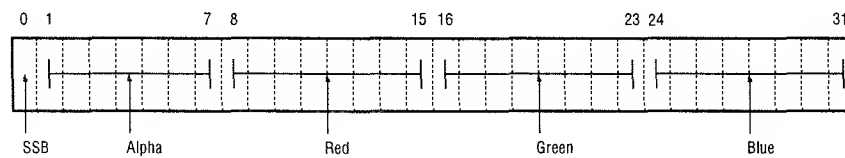
blendOp	Definition
0x0	LERP ($A*(1-t) + B * t$)
0x1	MULT ($A * B$)

Note: For the MULT function, the *t* input to the LERP block is don't care.

TAB Constant Registers (0x0004_6448 & 0x0004_644C)

The SSB from the output of the filter is used to select between these two registers. The output for SSB, alpha or color from the TAB can be independently controlled to be from the selected constant register or from the pipeline.

CLT_TXTCONSTn (pp, blue, green, red, alpha, ssb) n = 2, 3



Texture Loader Register Definitions

The texture loader registers are defined in the following paragraphs.

TXTCNST (0, 1, 2, 3) - Source Expansion Constant Registers (0x0004_6440 - 0x0004_644C)

These four sets of constants registers are used during the decompression of compressed textures to generate constant colors or index for each of the four source texture texel types. TXTCNST0 and TXTCNST1 select the PIP constant. TXTCNST2 and TXTCNST3 select the texture application constant. For indexed types, the *Blue* field can be used as an offset that is added to the input index value. If no index value is provided, then several bits (depending on the color depth) may be taken as an index constant.

Note that when 5 bits of color is selected for expansion, the 5-bit color is right-aligned into the 8-bit channel. Likewise, 4-bit alpha is right-aligned into the 7-bit alpha channel.

The SSB from the TRAM Lookup is used to select between these two registers. The output for SSB, alpha or color can be independently controlled to be from the selected register.

CLT_TXTCNSTn (pp, blue, green, red, alpha, ssb)

Field	Description
BLUE	Select Blue value
GREEN	Select Green value
RED	Select Red value
ALPHA	Select Alpha value
SSB	Select SSB value

TXTSRCTYPE n - Texture Source Types (0x0004_6434 - 0x0004_643c)

These register are used during texture load of compressed textures only. There are five format description registers—one for each of the four compressed texel types within the source texture (0, 1, 2, 3), and one for the expanded format that is used to be loaded into the TRAM.

CLT_TXTSRCTYPE n (pp, cdepth n , adepth n , istrans n , hassssbn, hascolor n , hasalphan, isliteral n)

Field	Description
CDEPTH(0, 1, 2, 3)	Number of bits per color component for literal formats (0x5=5 bits per color, 0x8=8 bits per color)
ADEPTH(0, 1, 2, 3)	Number of Alpha bits (0x4=4 bits Alpha, 0x7=7 bits)
ISTRANS(0, 1, 2, 3)	Selects constant Alpha or SSB for run of compressed texels
HASSSB(0, 1, 2, 3)	Specifies if SSB is present
HASCOLOR(0, 1, 2, 3)	Specifies if color is present for literal texel types
HASALPHA(0, 1, 2, 3)	Specifies if Alpha is present
ISLITERAL(0, 1, 2, 3)	If set, the type is a literal format

CLT_TXTEXPTYPE(pp, cdepth, adepth, istrans, hasssb, hascolor, hasalpha, isliteral)

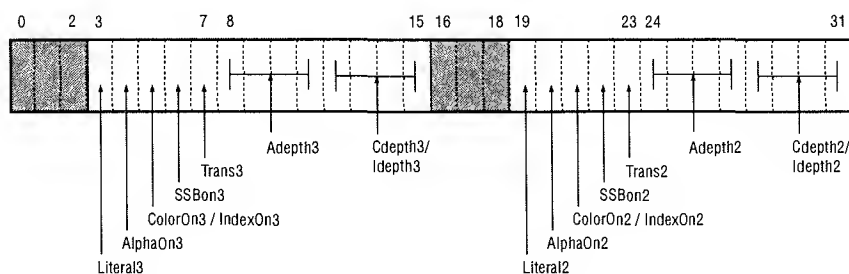
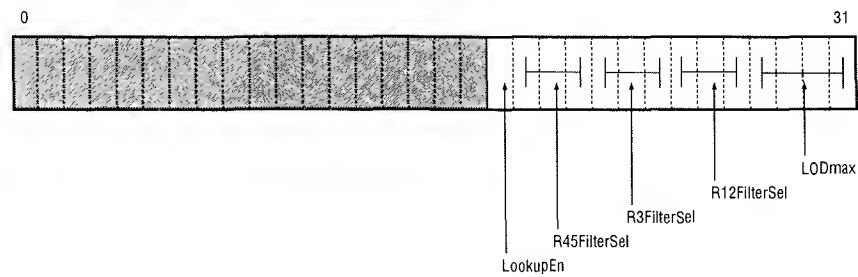
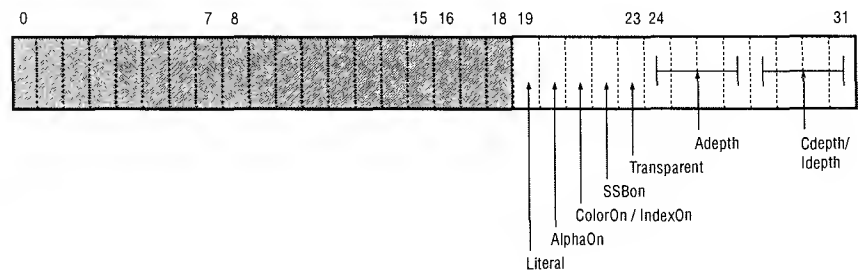


Figure 3-15 TxtLdSrcType01

Figure 3-16 *TxtLdSrcType23*Figure 3-17 *TxtExpType*

These registers are used during texture load of compressed textures only. There are five format description registers—one for each of the four compressed texel types within the source texture, and one for the expanded format that is used to be loaded into the TRAM.

The fields specify the following information:

- ◆ *Cdepth*—Literal formats are not supported in compressed textures.
- ◆ *Iddepth*—The number of bits of index for indexed formats. Only the following are supported:

Iddepth	Definition
0x0	reserved
0x1 - 0x8	1 - 8bits index
0x9 - 0xF	reserved

- ◆ *Adepth*—The number of bits of alpha. Only a few are valid:

Adepth	Definition
0x0 - 0x3	reserved
0x4	4 bits alpha
0x5 - 0x6	reserved

Adepth	Definition
0x7	7 bits alpha
0x8 - 0xF	reserved

- ◆ *Transparent*—Used only for compressed texture source description. If this field is set, the decompressor uses a constant color, alpha or SSB for that run length of compressed texels. The constant register is selected from one of the four *TxtSrcConstN* registers, where *N* indicates the type of the current control byte within the decompressor.
- ◆ *SSBon*—Specifies whether an SSB is present.
- ◆ *ColorOn, IndexOn*—Specifies whether index color is present. Note that there are combinations that have no color bits.
- ◆ *AlphaOn*—Specifies whether alpha is present.
- ◆ *Literal*—If this field is set, the type is a literal format. If this field is not set, the type is indexed.

The decompressor can support all the indexed pipeline formats. Literal pipeline formats can not be compressed. The texels are decompressed at different rates depending on their total depth. The following table shows the complete set of supported texels and the speed at which they can be loaded.

Color	Alpha	SSB	Total	Speed
0	-	1	1	4
1	-	-	1	4
1	-	1	2	4
2	-	-	2	4
0	4	-	4	4
3	-	1	4	4
4	-	-	4	4
6	-	-	6	2
5	-	1	6	2
1	4	1	6	2
2	4	-	6	2
0	7	1	8	2
3	4	1	8	2
4	4	-	8	2
7	-	1	8	2
8	-	-	8	2

Color	Alpha	SSB	Total	Speed
4	7	1	12	1
7	4	1	12	1
8	4	-	12	1
8	7	1	16	1

In the preceding table, the speed column specifies the number of texels loaded per tick. The speed is independent of the source type—it depends only on the expanded type to be written to the TRAM.

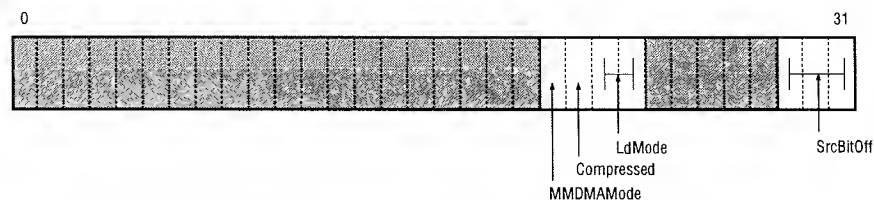
Note also that compressed source texels cannot contain more data for each of the data types (color, alpha, SSB) than the expanded format. They can only contain an equal amount or less (none).

TXTLDCNTL - Texture Loader Control Register (0x0004_6404)

This register controls the texture load process. When the TLD instruction is issued, this register is used to determine which type of load function to select.

CLT_TXTLDCNTL (pp, compressed, loadmode, srcbitoff)

Field	Description
COMPRESSED	Specifies that the source texture in memory is compressed and needs decompression before use.
LOADMODE LOADMODE_TEXTURE LOADMODE_MMDMA LOADMODE_PIP	Determines the operation of the memory-to-memory DMA process.
SRCBITOFF	Source bit offset used with uncompressed texture loading.



The bits shown in the preceding figure are defined as follows:

- ◆ *LdMode*—Specifies the mode of the texture loader.

LdMode[0:1]	Definition
0x0	Texture Load
0x1	MM DMA
0x2	PIP Load
0x3	reserved

Note: When MMDMA is selected, the target is determined by the MMDMATramOn and MMD-MAPipOn bits within the supervisor TxtCntl register.

- ◆ *Compressed*—If this bit is enabled, it indicates that the source texture in memory is compressed and must be decompressed before it is used. This bit is used only if *LdMode* = 0 (i.e., Texture Load).
- ◆ *SrcBitOff*—This field is used by the texture loader during the loading of uncompressed textures (*LdMode* = 0 & *Compressed* == 0). It indicates the bit start position within a byte. This is used in conjunction with *TxtLdSrcBase* to define the exact bit start position of a texel in memory.

TXTLDSRCOFFSET - Text Load Source Offset Register

This register provides the compressed texture start offset value.

CLT_TXTLDSRCOFFSET (somevalue)

Field	Description
VALUE	Select offset value

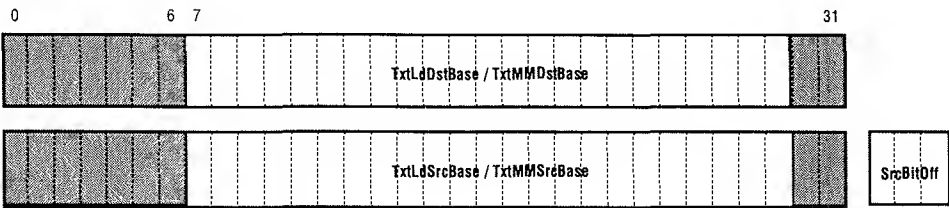
TXTLDSRCADDR- Loader Base Registers(0x0004_6424 & 0x0004_6414)

TxtLdSrcBase describes the byte-aligned source address in main memory. Compressed textures may start on any byte aligned address. Uncompressed textures may start on any bit address. The *SrcBitOff* field within the *TxtLdCntl* register is effectively used as three extra MSBs for uncompressed load. PIP contents are byte aligned as well as MMDMA source addresses.

CLT_TXTLDSRCADDR(pp, x)

TxtLdDstBase describes the 32-bit word aligned destination address within the TRAM. The base address of all load functions will be on a 32-bit boundary. This is because the TRAM logic does not support a RMW cycle into the RAM to allow for arbitrary alignment

CLT_TXTLODBASE0(pp, x)



TXTCOUNT - Texture Loader Count Register (0x0004_6428)

This register specifies different counts for the various loader modes:

- ◆ During uncompressed texture load, TxtLdRowCnt indicates the number of rows of texels to copy into the TRAM.
- ◆ During compressed texture load, TxtLdTexCnt indicates the total number of texels to decompress.
- ◆ During MMDMA, TxtLdByteCnt indicates the number of bytes to copy.
- ◆ During PIP load, TxtLdByteCnt indicates the number of bytes to copy into the PIP.

CLT TXTCOUNT (somevalue)

Field	Description
TXTCOUNT	<p>During compressed texture load, indicates the total number of texels to decompress.</p> <p>During compressed texture load, indicates the total number of texels to decompress.</p> <p>During MMDMA, indicates the number of bytes to copy.</p> <p>During PIP load, indicates the number of bytes to copy into the PIP.</p>

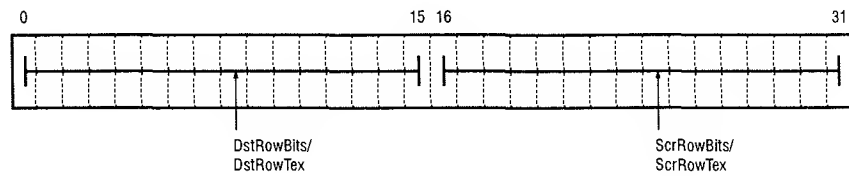
Note that a count of 0 does mean zero—that is, “Do something zero times.”



TxtUVMax-Texture Loader Width Register (0x0004_642C)

During texture load of uncompressed textures, this register indicates the width of the destination and source buffers in bits. A count of 0 indicates a width of zero.

CLT_TXTUVMAX(pp, umax, vmax)

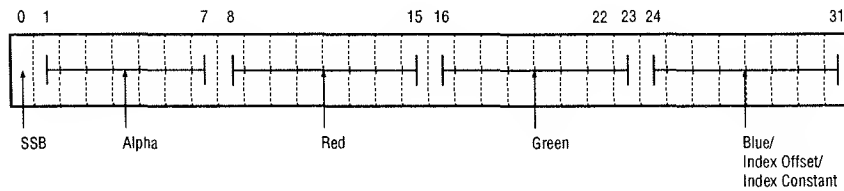


TxtConst2-3-Decompress Constant Registers (0x0004_6448 - 0x0004_644C)

M2 provides two constant registers. They are used only during decompresses of compressed textures. They are used to generate constant colors or index for each of the four source texture texel types. For indexed types the *Blue* field can be used as an offset which is added to the input index value. If no index value is provided, then several bits (depending on the color depth) may be taken as an index constant.

CLT_TXTCONSTn(pp, red, green, blue, alpha, ssb) n = 2,3

Note that when 5 bits of color are selected for expansion, the 5-bit color is right-aligned into the 8-bit channel. Likewise, 4-bit alpha is right-aligned into the 7-bit alpha channel.



Destination Blender

Once pixels have been passed through the Texture Mapper, they pass into the Destination Blender, where the last four processing steps take place:

- Pixels can be optionally discarded from a triangle, based on several criteria.
- Pixels can be blended with a color, or pixels from another frame buffer.
- Pixels can be dithered.
- Pixels can be Z-buffered, so that they are only drawn if they are from visible portions of a triangle that intersects other triangles.

Finally, the pixels are written to the frame buffer.

This chapter contains the following topics:

Topic	Page Number
Pixel Discards	60
Pixel Blending	60
Pixel Scaling	61
Source Blending	61
Dithering	62
Z-buffering	62
Window Clipping	63
Destination Blender Register Definitions	63

Pixel Discards

In the Destination Blender, pixels can be discarded before being output to the frame buffer or Z-buffer. This can be useful, for example, to mask away unwanted parts of a triangle that fall outside of the shape of a sprite that is texture-mapped onto a pair of triangles. Discards can be set to happen whenever any combination of the following events occur:

- Alpha of the pixel is 0.0
- Color of the pixel is pure black (red=0.0, green=0.0, blue=0.0)
- SSB of the pixel is 0
- Pixel falls outside the area covered by the Z-buffer

Note: *Since the pixels enter the Destination Blender after being passed through the Texture Mapper, the color, alpha, and SSB used for discards are the ones that come from the Texture Blender.*

The Triangle Engine only allows you to allocate a Z-buffer that is smaller than the frame buffer. The Z-discard allows pixels that fall outside of the Z-buffered area to be immediately discarded. For more information, see the “Z-buffering” section, later in this chapter.

There is no performance hit when discards are enabled, and they can allow better performance in some cases, since discarded pixels are not passed into the Z-buffering unit or pixel blender.

Pixel Blending

After the discard process is complete, the remaining pixels are passed to a sophisticated blender, where the color can be blended with a color from any of a number of sources, including a second source frame buffer. This blender is typically used to make pixels partially transparent, or to globally tint a scene toward a constant color, such as in a screen fade. Because the blend can be weighted based on alpha, carefully constructed triangles can use the Destination Blender to apply fog and some other special effects to the scene.

The two colors to be blended (we’re calling them A and B) can be blended according to the following equation:

$$C_{out} = ((MA \bullet A) \text{ op } (MB \bullet B)) \ll \text{shift}$$

A, B, MA, and MB can come from a variety of sources. The following table describes the different combinations that can be used for these variables:

Variable	Possible Sources
A	Color of the pixel from the Texture Mapper Constant color Alpha of the pixel from the Texture Mapper Complement of the color from the source frame buffer

Variable	Possible Sources
B	Color from a source frame buffer Constant color Alpha of the pixel from the Texture Mapper Complement of the color of the pixel from the Texture Mapper
MA	Alpha of the pixel from the Texture Mapper Alpha from the source frame buffer One of a pair of constants (selected by the SSB) Color of the pixel from the source frame buffer
MB	Alpha of the pixel from the Texture Mapper Alpha from the source frame buffer One of a pair of constants (selected by the SSB) Color of the pixel from the Texture Mapper

Besides the flexibility that the various selectors for A, B, MA, and MB provide, you can also decide what operation you want to perform. The operation can be an add or subtract, with or without clamping, or any boolean operator, such as AND, OR, etc.

Finally, color components can be bit-shifted left or right up to 3 bits. The shift is actually always a left shift, but the parameter allows for a shift left by -3 bits, which results in a right shift 3 bits.

When the pixel blender is enabled, pixels can be output at a maximum throughput of 1 pixel per tick.

Pixel Scaling

The Triangle Engine performs color calculations on 8-bit numbers for each channel. Source images, as well as destination frame buffers, can be 16- or 32-bit numbers. The Triangle Engine converts the 16-bit pixel values to 32-bit values before any computations are performed.

You have an option to right-shift by three bits the input source and/or texture 8-bit values. In this way, your application can combine as many as eight images without overflowing the pixel values. You can use this capability to accumulate images in the destination frame buffer. Resulting accumulated images are then scaled appropriately. Attributes associated with this capability are the `DblARightJustify` attribute, the `DblBRightJustify` attribute, and the `DblFinalDivide` attribute.

Source Blending

By blending pixels with a source frame buffer, you can easily produce effects like transparency and reflections. But, you need to set up the correct registers in order to ensure correct output. The M2 Triangle Engine needs to know the source frame buffer's address, width, height, pixel depth, and stride. The source frame buffer can be the same bitmap that is used as the destination frame buffer.

When blending with a source frame buffer is enabled, pixels can be output at a maximum rate of 1 pixel every other tick.

Dithering

The M2 Triangle Engine can dither pixels before they are sent to the destination frame buffer. This reduces the visual artifact of mock banding when 16-bit frame buffers are used. Here's how the ditherer works:

You load a 4 x 4 matrix of dither values. Valid matrix values are 4-bit signed integers, between -8 and +7. The matrix is packed into two 32-bit registers. One register contains the matrix values for the top two rows, the other the bottom two rows.

The least significant two bits of the x- and y-coordinates of the pixel are used as indices into the 4 x 4 matrix. The value from the matrix is retrieved, and added to the red, green, and blue components of the pixel.

Dithering can be performed at a rate of 2 pixels per tick.

See Appendix A for the register setup used with dithering.

Z-buffering

The M2 Triangle Engine supports Z-buffering for pixels that contain depth information (triangles which have a W value). By using a Z-buffer, you can render objects in any order, and have them intersect each other, without worrying about how overlapping regions are drawn. Normally, the Z-buffer only draws pixels that are closer to the viewer than previously-rendered pixels, but it can be set to work in other modes.

The Z-buffer works by storing depth information in a special 16-bit frame buffer. The contents of each 16-bit pixel are in a proprietary 3DO format. At the beginning of a frame, the best way to clear the Z-buffer is to render two large triangles with a W value of 0.0, and set the Z-bufferer to force updates to the Z-buffer, regardless of what the value is. For more information about modifying the operating mode of the Z-buffer, see the description of the DBZCNTL register in the "Destination Blender Register Definitions" section, later in this chapter.

It is not necessary to allocate a Z-buffer to be the size of the entire frame buffer. In some cases, such as an application where the top portion of the screen is used for player's status, it might only be necessary to allocate a Z-buffer to be the size of the playfield area of the screen, thus saving some memory.

Note: *Although the Z-buffer does not need to be as tall as the destination frame buffer, it must be as wide as the destination frame buffer's stride.*

Z-banding

One other technique that can save some RAM (but at the cost of some performance) is to allocate a Z-buffer that is a fraction of the height of the frame buffer. This technique is referred to as *Z-banding*. Here's how Z-banding works: The Z-buffer is aligned with the top of the destination frame buffer. The scene is rendered and the Z-buffer is moved down. The scene is re-rendered and the Z-buffer moved down again. This process is repeated until the whole frame buffer is drawn. Using this technique, in combination with the Destination Blender's discard when pixels are outside the region covered by the Z-buffer, allows Z-buffering to occur when there is not enough RAM for an entire Z-buffer.

However, because the Triangle Engine needs to pass over the data multiple times, performance will usually not be as good as the case where a complete Z-buffer can be allocated.

When Z-buffering is enabled, performance is 2 pixels per tick if the pixels do not need to update the frame buffer or Z-buffer, and 1 pixel per tick in the case where the pixel needs to be drawn.

See Appendix A for the register setup used with Z-buffering.

Window Clipping

An application can specify a rectangular window for clipping out the pixels that lie on the inside or the outside of this rectangle. The rectangle should be specified in conjunction with the `DblEnableAttrs` attributes of `WinClipInEnable` or `WinClipOutEnable`. The *wWindow* rectangle is specified with `DblXWinClipMin`, `DblXWinClipMax`, `DblYWinClipMin`, and `DblYWinClipMax`.

Destination Blender Register Definitions

This section describes the individual registers used to control the Destination Blender. Each command register can be broken down into fields. For a register named XXX, the macro `CLA_XXX` provides the data word to be written to the register with the arguments converted and packed into the register fields. The macro `CLT_XXX` takes a pointer to a pointer as the first argument. It writes the argument word to the command list and advances the pointer.

This section provides descriptions of each of the registers, the address of the register, the CLT macro to set the register and the register layout.

Warning: *Some registers are used for different purposes during the rendering and loading stages.*

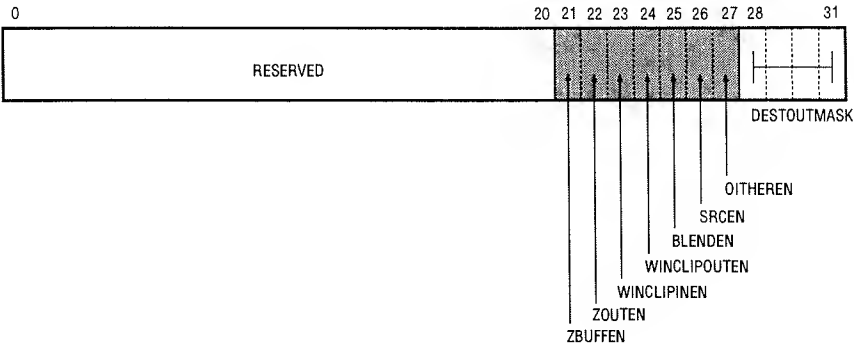
DBUSERCONTROL - User Destination Blend Control Register (0x0004_8008)

This register provides general control of the destination blend logic.

```
CLT_DBUSERCONTROL(pp, zbuffen, zouten, winclipinen, winclipouten,
                  blenden, srcen, ditheren, destoutmask)
```

Field	Description
ZBUFFEN	Enable Z-buffering.
ZOUTEN	Enable Z-buffer output.
WINCLIPINEN	Enable window clipping inside the clip range.
WINCLIPOUTEN	Enable window clipping outside the clip range.
BLENDEEN	Enable color, Alpha, and DSB blending.
SRCEN	Master control for the source input. Enables color, Alpha, and DSB blending.

Field	Description
DITHEREN	Enable dithering for output to 555.
DESTOUTMASK	Enable byte output (DSB, RGB, Alpha)

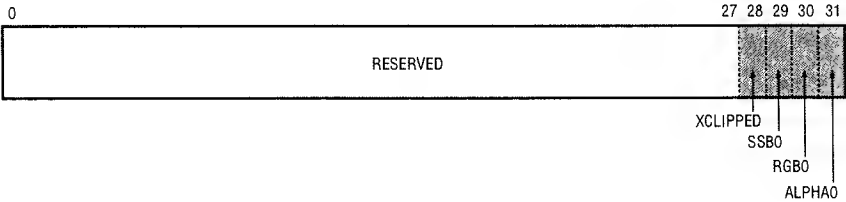


DBDISCARDCONTROL - Pixel Discard Control Register (0x0004_800c)

This register provides pixel discard control.

CLT_DBDISCARDCONTROL (pp, zclipped ssb0, rgb0, alpha0)

Field	Description
ZCLIPPED	Enable pixel discard when outside of the buffer region.
SSB0	Enable pixel discards based on SSB.
RGB0	Enable pixel discards if R == G == B == 0.
ALPHA0	Enable pixel discards if Alpha == 0.



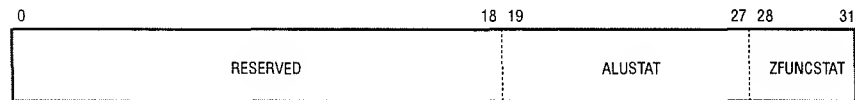
DBINTCNTRL - Interrupt Control Register (0x0004_8014)

This register provides masks to control which ALU and Z results can generate

interrupts to the CPU.

CLT_DBINTCNTL (PP, ALUSTAT, ZFUNCSTAT)

Field	Description
ALUSTAT	AND'd with ALU status bits and OR'd to create ALUStatInt
ZFUNCSTAT	AND'd with Z function status bits and OR'd to create ZFuncInt

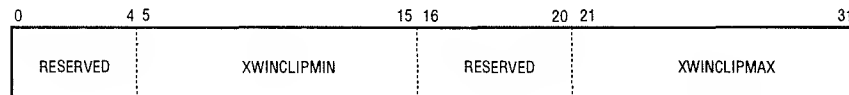


DBXWINCLIP - X Window Clip Value Register (0x0004_801c)

This register defines the X border of the user-defined render window.

CLT_DBXWINCLIP (pp, xwinclipmin, xwinclipmax)

Field	Description
XWINCLIPMIN	X window clip minimum (left)
XWINCLIPMAX	X window clip maximum (right)

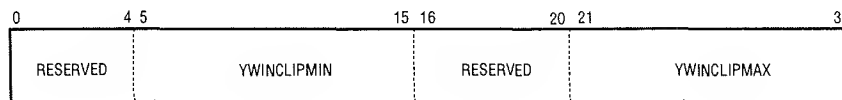


DBYWINCLIP - Y Window Clip Value Register (0x0004_8020)

This register defines the Y border of the user-defined render window.

CLT_DBXWINCLIP (pp, xwinclipmin, xwinclipmax)

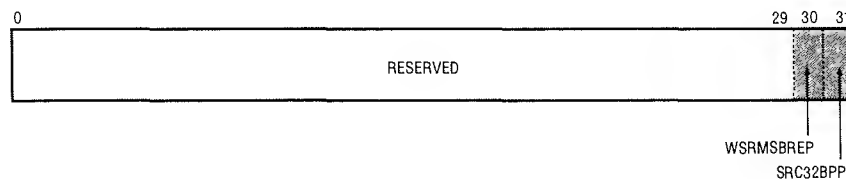
Field	Description
YWINCLIPMIN	Y window clip minimum (left)
YWINCLIPMAX	Y window clip maximum (right)

**DBSRCCNTL - Source Read Control Register (0x0004_8030)**

Specifies the format of the buffer to be used for source blending.

CLT_DBSRCCNTL (pp, srcmsbred, src32bpp)

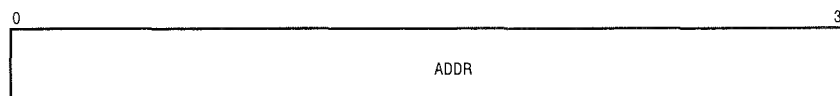
Field	Description
WSRMSBREP	For 16 Bpp, Replicate 3 MSBs into 3 LSB's if left justified (otherwise insert zeros).
SRC32BPP	Source pixel format (16 or 32 bpp).

**DBSRCBASEADDR - Source Base Address Register (0x0004_8034)**

This register provides the source bitmap address when the destination blender's source blending mode is enabled.

CLT_DBSRCADDR (pp, addr)

Field	Description
ADDR	Base address of source buffer

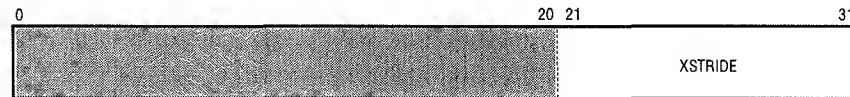
**DBSRCXSTRIDE - Source X Stride Register (0x0004_8038)**

This register contains the source read X stride value - the distance in pixels to the

next line.

```
CLT_DBSRCXSTRIDE (pp, xstride)
```

Field	Description
XSTRIDE	X Stride value.

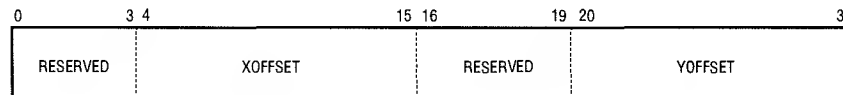


DBSRCOFFSET - Source Offsets Register (0x0004_803c)

This register enables the X and Y offsets used for source reads.

```
CLT_DBSRCOFFSET (pp, xoffset, yoffset)
```

Field	Description
XOFFSET	X offset
YOFFSET	Y offset

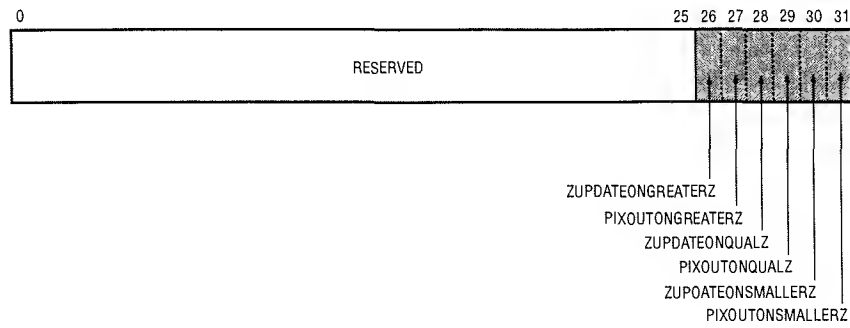


DBZCNTRL - Z Buffer Control Register (0x0004_8040)

This register defines the ZPixOut and ZBufOut results.

```
CLT_DBZCONTROL (pp, zupdateongreaterz, pixoutongreaterz,
                zupdateonqualz, pixoutonqualz, zupdateonsmallerz,
                pixoutonsmallerz)
```

Field	Description
ZUPDATEONGREATERZ	If new Z greater than current Z, update Z
PIXOUTONGREATERZ	If new Z greater than current Z, update pix
ZUPDATEONQUALZ	If new Z equal to current Z, update Z
PIXOUTONQUALZ	If new Z equal to current Z, update pix
ZUPDATEONSMALLERZ	If new Z smaller than current Z, update Z
PIXOUTONSMALLERZ	If new Z smaller than current Z, update pix

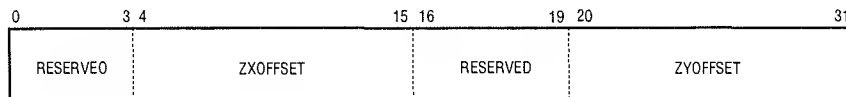


DBZOFFSET - Z Offset Register (0x0004_8048)

This register contains the X and Y offsets used for Z.

CLT_DBZOFFSET (pp, zxoffset, zyoffset)

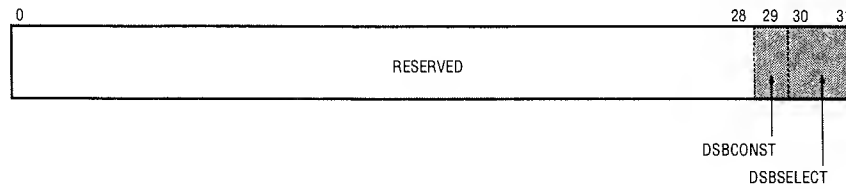
Field	Description
ZXOFFSET	X offset for Z
ZYOFFSET	Y offset for Z



DBSSBDSBCNTL - SSB/DSB Control Register (0x0004_8050)

CLT_DBSSBDSBCNTL (pp, dsconst, dbselect_xxxxxxx)

Field	Description
DSBCONST	DSB constant
DSBSELECT_	Select DSB generation:
OBJSSB	Use SSB (0)
CONST	Use constant (1)
SRCSSB	Use source input (2)

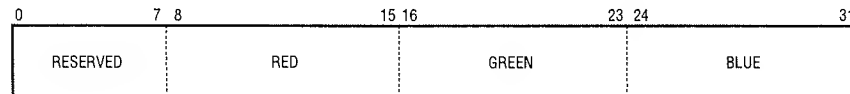


DBCONSTIN - Constant In Register (0x0004_8054)

This register contains the RGB constant used for computation input.

CLT_DNCONSTIN (pp, red, green, blue)

Field	Description
RED	Red constant
GREEN	Green constant
BLUE	Blue constant



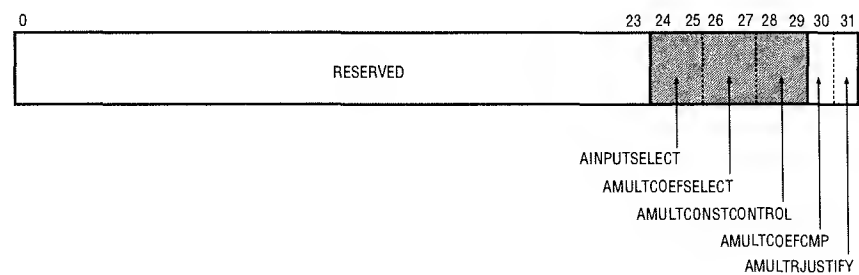
DBAMULTCNTL - Texture Multiplication Control Register (0x0004_8058)

This register controls the texture color multiplication.

CLT_DBAMULTCNTL (ainputselect, amultcoefselect,
amultconstcontrol, amultrjustify)

Field	Description
AINPUTSELECT	Choose texture, constant, or 1-Cs for texture input: TEXCOLOR (0) CONSTCOLOR (1) SRCCOLORCOMPLEMENT (2) TEXALPHA (3)

Field	Description
AMULTCOEFSELECT	Choose texture coefficient: TEXALPHA (00) SRCALPHA (08) CONST (10) SRCCOLOR (18) TEXALPHACOMPLEMENT (01) SRCALPHACOMPLEMENT (09) CONSTCOMPLEMENT (11) SRCCOLORCOMPLEMENT (19)
AMULTCONSTCONTROL	Choose constant controller: TEXSSB (0) SRCSSB (1)
AMULTCOEFCMP	Use (1-coef) for texture coefficient
AMULTRJUSTIFY	Right shift 888 texture values to 555

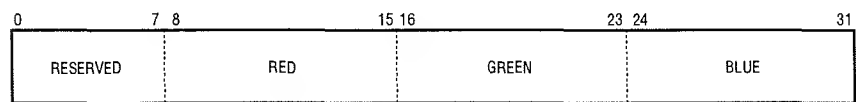


DBAMULTCONSTSSB0 - Texture Coefficient Constant 0 Register (0x0004_805c)

This register contains the texture RGB coefficient constant “0.”

CLT_DBAMULTCONSTSSB) (pp, red, green, blue)

Field	Description
RED	Red constant 0
GREEN	Green constant 0
BLUE	Blue constant 0

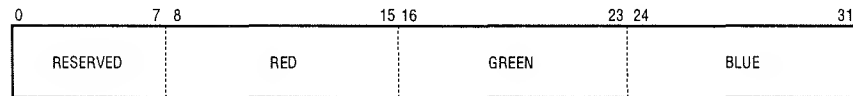


DBAMULTCONSTSSB1 - Texture Coefficient Constant 1 Register (0x0004_8060)

This register contains the texture RGB coefficient constant "1."

CLT_DBAMULTCONSTSSB1 (pp, red, green, blue)

Field	Description
RED	Red constant 1
GREEN	Green constant 1
BLUE	Blue constant 1

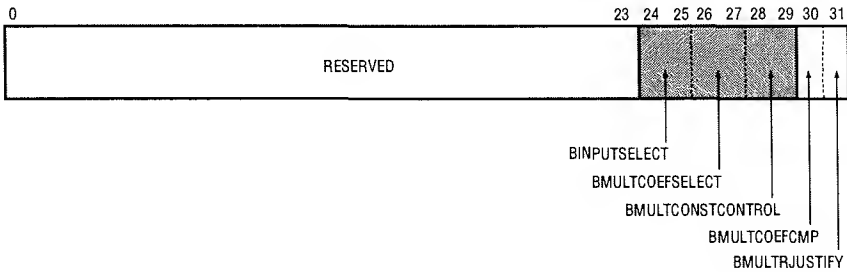
**DBBMULTCNTL - Source Multiplication Control Register (0x0004_8064)**

This register provides the source multiplication control.

CLT_DBBMULTCNTL (pp, binputselect, bmultcoefselect, bmultconstcontrol, bmulttrjustify)

Field	Description
BINPUTSELECT	Choose source or constant for source input: SRCCOLOR (0) CONSTCOLOR (1) TEXCOLORCOMPLEMENT (2) SRCALPHA (3)
BMULTCOEFSELECT	Choose source coefficient: TEXALPHA (00) SRCALPHA (08) CONST (10) TEXCOLOR (18) TEXALPHACOMPLEMENT (01) SRCALPHACOMPLEMENT (09) CONSTCOMPLEMENT (11) TEXCOLORCOMPLEMENT (19)
BMULTCONSTCONTROL	Choose constant controller: TEXSSB (0) SRCSSB (1)

Field	Description
BMULTCOEFCMP	Use (1-coef) for source coefficient
BMULTRJUSTIFY	Right shift 888 source values to 555

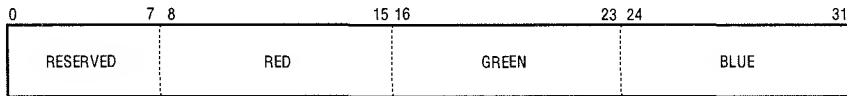


DBBMULTCONSTSSB0 - Source Coefficient Constant 0 Register (0x0004_8068)

This register contains the source RGB coefficient constant “0.”

CLT_DBBMULTCONSTSSB0 (pp, red, green, blue)

Field	Description
RED	Red constant 0
GREEN	Green constant 0
BLUE	Blue constant 0

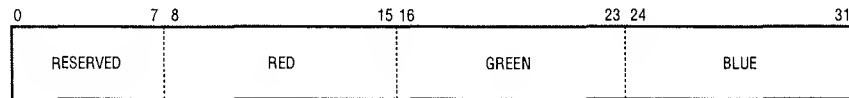


DBBMULTCONSTSSB1 - Source Coefficient Constant 1 Register (0x0004_806c)

This register contains the source RGB coefficient constant “1.”

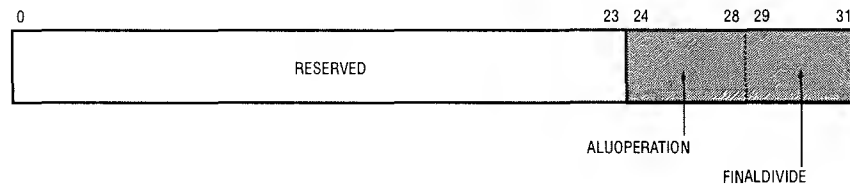
CLT_DBBMULTCONSTSSB1 (pp, red, green, blue)

Field	Description
RED	Red constant 1
GREEN	Green constant 1
BLUE	Blue constant 1

**DBALUCNTL - ALU Control Register (0x0004_8070)**

CLT_DBALUCNTL (pp, aluoperation, finaldivide)

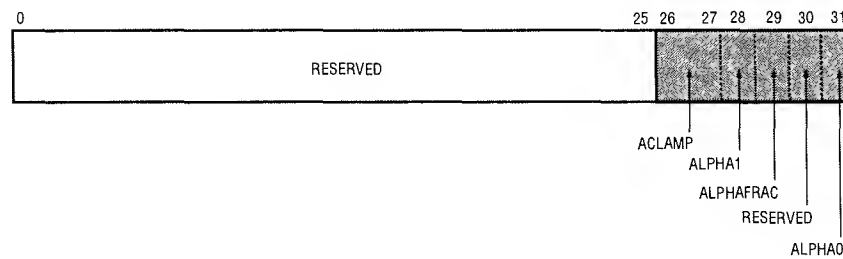
Field	Description
ALUOPERATION	ALU selection: A_PLUS_BCLAMP (0) A_PLUS_B (2) A_MINUS_BCLAMP (8) A_MINUS_B (a) B_MINUS_ACLAMP (c) B_MINUS_A (e) OUTPUTZERO (10) NEITHER (11) NOTA_AND_B (12) NOTA (13) NOTB_AND_A (14) NOTB (15) XOR (16) NOTA_AND_B (17) A_AND_B (18) ONEONEQUAL (19) B (1a) NOTA_OR_B (1b) A (1c) NOTB_OR_A (1d) A_OR_B (1e) OUTPUTONE (1f)
FINALDIVIDE	Final shift 0, 1, or 2



DBSRCALPHACNTL - Source Alpha Control Register (0x0004_8074)

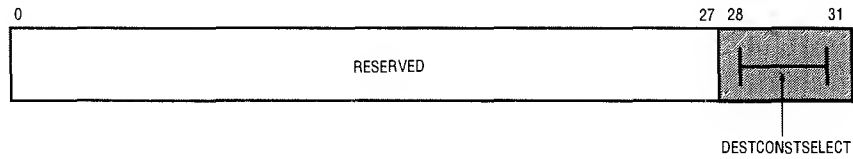
CLT_DBSRCALPHACNTL (pp, aclamp, alpha1, alphafrac, alpha0)

Field	Description
ACLAMP	Specify clamping option for three alpha ranges (3 @ 2 bit): LEAVEALONE (0) FORCE1 (1) FORCE0 (2)
ALPHA1	Two-bit field for Alpha equals 0xff
ALPHAFRAC	Two-bit field for fractional Alpha
ALPHA0	Two-bit field for Alpha equals 0

**DBDESTALPHACNTL - Destination Alpha Control Register (0x0004_8078)**

CLT_DBDESTALPHACNTL (pp, destconstselect)

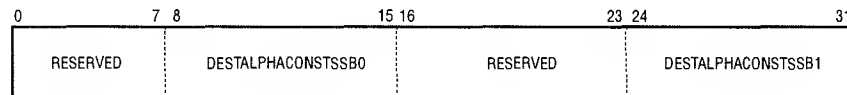
Field	Description
DESTCONSTSELECT	Combines meta select and select: TEXALPHA (0) TEXSSBCONST (1) SRCSSBCONST (9) SRCALPHA (2) RBLEND (3)



DBDESTALPHACONST - Destination Alpha Constants Register (0x0004_807c)

CLT_DBDESTALPHACONST (pp, destalphaconstssb0, destalphaconstssb1)

Field	Description
DESTALPHACONSTSSB0	Destination Alpha constant 0
DESTALPHACONSTSSB1	Destination Alpha constant 1



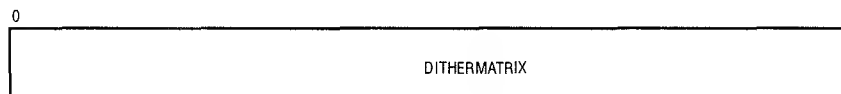
DBDITHERMATRIX - Dither Matrix Registers (0x0004_8080, 0x0004_8084)

This register contains the 4x4 dither matrix. The register at 0x0004_8080 holds the top two rows of the matrix, and is accessed using the `dmA` argument.

The register at 0x0004_8084 holds the bottom two rows of the matrix, and is accessed using the `dmB` argument.

CLT_DBDITHERMATRIX (pp, dmA, dmB)

Field	Description
X0Y0 - X3Y0/X0Y1 - X3Y1	First half of 4x4x4 dither matrix
X0Y2 - X3Y2/X0Y3 - X3Y3	Second half of 4x4x4 dither matrix

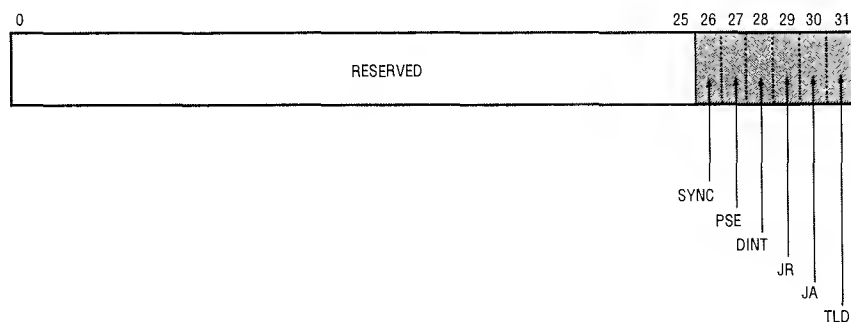
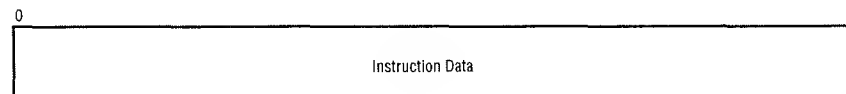


DCNTL - Deferred Flow Control Register (0x0004_0010, 0x0004_0014)

This register pair is used to perform various flow control instructions (see the "Flow Control Instructions" section, in Chapter 1, for more information). The data register is first loaded with the instruction data. The instruction register is then loaded with the instruction itself:

```
CLT_Sync (pp)
CLT_Pause (pp)
CLT_Interrupt (pp)
CLT_JumpRelative (pp)
CLT_JumpAbsolute (pp)
CLT_TxLoad (pp)
```

Field	Description
SYNC	Pause instruction execution until the entire TE pipe has been flushed
PSE	Pause execution
DINT	Interrupt CPU with vector - TEDCntlData is used as vector data
JR	Jump relative - use TEDCntlData as (2's complement) offset
JA	Jump absolute - use TEDCntlData as destination address
TLD	Texture load

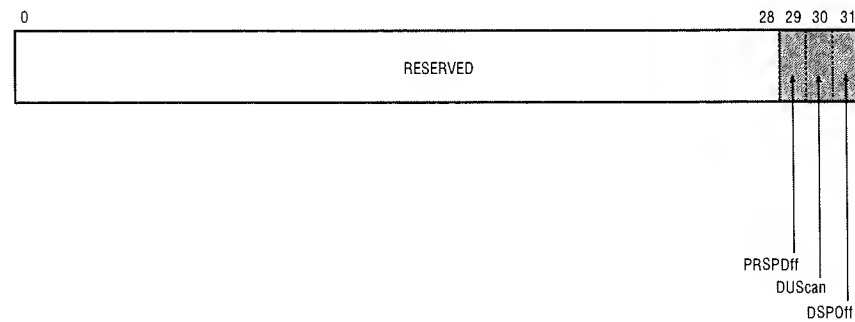


ESCNTL - Edge Walker/Span Walker Control Register (0x0004_4000)

This register contains three control bits. One turns perspective on and off, the second controls whether triangles are processed (scanned) from top-to-bottom or bottom-to-top, and the third turns DoubleStrike Prevention on and off.

CLA_ESCNTL (perspective, duscan, dspoff)

Field	Description
PRSPOff	Turn perspective calculations off.
DUScan	Down/up scan direction flags.
DSPOff	Disables Double Strike Prevention

**Register Memory Map**

Address	Name	Format	Access	Description
0x0004_8008	DBUserControl	Bit	RWSC	User Destination Blend Control
0x0004_800C	DBDiscard Control	Bit	RWSC	Pixel discard control
0x0004_8010	DBStatus	Bit	RWSC	Status indication
0x0004_8014	DBIntCntl	Bit	RWSC	Mask indicating which ALU and Z results should generate interrupts.
0x0004_801C	DBXWinClip	U_Fix	RW	X window clip values
0x0004_8020	DBYWinClip	U_Fix	RW	Y window clip values

Address	Name	Format	Access	Description
0x0004_8030	DBSrcCntl	Bit	RWSC	Source read Control register
0x0004_8034	DBSrcBaseAddr	Addr	RW	Source read base pointer
0x0004_8038	DBSrcXStride	U_Fix	RW	Source read X Stride value - Distance in pixels to next line.
0x0004_803C	DBSrcOffset	S_Fix	RW	X and Y Offset used for Source reads
0x0004_8040	DBZCntl	Bit	RWSC	Z Buffer Control register
0x0004_8048	DBZOffset	S_Fix	RW	X and Y Offset used for Z.
0x0004_8050	DBSSBDSBCntl	Bit	RWSC	SSB & DSB Control register
0x0004_8054	DBConstIn	U_Fix	RW	Const RGB used for computation input
0x0004_8058	DBAMultCntl	Bit	RWSC	Texture color multiplication control
0x0004_805C	DBAMultConst SSB0	U_Fix	RW	Texture RGB coef const "0"
0x0004_8060	DBAMultConst SSB1	U_Fix	RW	Texture RGB coef const "1"
0x0004_8064	DBBMultCntl	Bit	RWSC	Texture color multiplication control
0x0004_8068	DBBMultConst SSB0	U_Fix	RW	Source RGB consts "0"
0x0004_806C	DBBMultConst SSB1	U_Fix	RW	Source RGB consts "1"
0x0004_8070	DBALUCntl	Bit	RWSC	ALU Control register
0x0004_8074	DBSrcAlphaCntl	Bit	RWSC	Control of source alpha
0x0004_8078	DBDest AlphaCntl	Bit	RWSC	Control of destination alpha
0x0004_807C	DBDestAlpha-Const	U_Fix	RW	Destination Alpha constants

Address	Name	Format	Access	Description
0x0004_8080	DBDither MatrixA	Bit	RWSC	First half of 4x4x4 dither ma- trix
0x0004_8084	DBDither MatrixB	Bit	RWSC	Second half of 4x4x4 dither ma- trix
0x0004_808C- 0x0004_9FFC	reserved		NA	

Note: Registers are not initialized at reset unless stated otherwise. They must be explicitly set by the OS or user. Once a register has been set, it will retain its value throughout all rendering until it is altered by a list instruction or the CPU directly. Note that the triangle engine must go through a sync operation when a control register is changed. Changing registers "on the fly" will produce unpredictable results and in some circumstances may hang the triangle engine until it is reset.

Register Setups

This appendix lists the Triangle Engine register configurations for the following operations:

- ◆ Z-buffering
- ◆ Dithering
- ◆ Transparencies
- ◆ Texturing (on/off)
- ◆ Perspective (on/off)
- ◆ Tiling a texture
- ◆ Loading an uncompressed texture
- ◆ Loading a PIP table

Z-Buffering

Turn Z-buffering on:

```
CLT_SetRegister(pp,DBUSERCONTROL,CLA_DBUSERCONTROL(1,1,0,0,0,0,0,0))
```

Select which pixels are removed by the Z-buffer (only pixels that are closer than what is in the Z-buffer will be drawn):

```
CLT_DBZCNTL(ptr, 0,0,0,0,1,1)
```

Set the Z offset:

```
CLT_DBZOFFSET(ptr,0,0)
```

You will be unable to do Z-buffering unless you've allocated a Z-buffer and told the TE about it (typically with `GS_SetZBuffer`). You will also have to clear your Z-buffer every frame, and include perspective values (`w`) with your triangles.

Dithering

Turn dithering on:

```
CLT_SetRegister(pp,DBUSERCONTROL,CLA_DBUSERCONTROL(0,0,0,0,0,0,1,0))
```

Set the dithering matrix to a basic dither pattern (see Figure A-1):

```
CLT_DbDitherMatrix(ptr,0xC0D12E3F,0xE1D0302F)
```

$$\begin{bmatrix} -4 & 0 & -3 & 1 \\ 2 & -2 & 3 & -1 \\ -2 & 1 & -3 & 0 \end{bmatrix}$$

Figure E-1 *Dithering matrix*

Setting Up the Destination Blender Source Buffer

To set up the frame buffer as a secondary source, set the following registers:

DBUSERCONTROL can be used to turn blending on, and to enable blending with a source frame buffer.

DBSRCBASEADDR points to the address of the frame buffer bitmap

DBSRCCNTL specifies whether the frame buffer is 16 or 32 bits

DBSRCOFFSET specifies an X and Y offset to use before reading from the frame buffer. This should usually be set to (0,0)

DBSRCXSTRIDE sets the width of the source frame buffer

Transparencies

There are a number of ways to do a transparency:

- To cause black (RGB 0,0,0) pixels, pixels with SSB 0, or pixels with Alpha 0 to be totally transparent, simply use the DBDISCARDCONTROL register. (Note: this checking takes place after all texturing has been done.)
- To set a constant amount of transparency (for instance, 90%), first set up the frame buffer as a secondary source (see above). Then call:

```
CLT_DBAMULTCNTL(ptr,  
RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,  
RC_DBAMULTCNTL_AMULTCOEFSELECT_CONST, 0, 0) to set the first source  
to come from the triangle data
```

```
CLT_DBBMULTCNTL(ptr, RC_DBBMULTCNTL_BINPUTSELECT_SRCCOLOR,  
RC_DBBMULTCNTL_BMULTCOEFSELECT_CONST, 0, 0) to set the second  
source to come from the source data
```

```
CLT_DBALUCNTL(ptr, RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0) to  
set the final math to add the two sources together
```

```
CLT_DBAMULTCONSTSSB0(ptr, 255-TRANS, 255-TRANS, 255-TRANS)
```

```
CLT_DBAMULTCONSTSSB1(ptr, 255-TRANS, 255-TRANS, 255-TRANS)
```

```
CLT_DBBMULTCONSTSSB0(ptr, TRANS, TRANS, TRANS)
```

CLT_DBBMULTCONSTSSB1(ptr, TRANS, TRANS, TRANS) to set up the degree of transparency, where TRANS is between 0 and 255, with 0 meaning totally opaque and 255 meaning totally transparent

- To set transparency to depend on the Alpha channel, first set up the frame buffer as a secondary source (see above). Then call:

```
CLT_DBAMULTCNTL(ptr,
RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,
RC_DBAMULTCNTL_AMULTCOEFSELECT_TEXALPHA, 0, 0);

CLT_DBBMULTCNTL(ptr, RC_DBBMULTCNTL_BINPUTSELECT_SRCCOLOR,
RC_DBBMULTCNTL_BMULTCOEFSELECT_TEXALPHACOMPLEMENT, 0, 0);

CLT_DBALUCNTL(ptr, RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0)
```

Then an alpha value of 1.0 will result in a completely opaque triangle, while an alpha of 0.0 will result in a completely transparent triangle, etc.

Texturing

To turn texturing on and off, use the lookup enable bit of the TXTADDRCNTL register. If you leave this bit set, you may get a performance hit even if you are drawing untextured triangles. However, there are a number of other registers that must be set before texturing can be used (not to mention that a texture load must be done before a texture can be drawn):

TXTADDRCNTL must also be used to select a filtering mode and set the number of LODs present.

TXTPIPCNTL must be set, even if you're using a literal (unindexed) texture. For a literal texture, it should be set so that the SSB, alpha and color come from the texture cache. For an indexed texture, the color should presumably come from the PIP, and the alpha and SSB can come from wherever you like. For indexed textures, this register also contains the PIP index offset, allowing you to select which part of the PIP will be used.

TXTEXTYPE must be set. This specifies what type of texture is being used (how many bits of color, whether it is indexed or unindexed, etc.) This will typically be set during your texture load

TXTUVMASK must be set. This register is used to tile textures (see below), but for untiled textures it should be set to (0x3ff, 0x3ff)

TXTUVMAX must be set. This sets the size of the texture, and will typically be set during your texture load. If multiple LODs are present, this register refers to the size of the smallest, and thus coarsest, LOD. Note that the values contained in this register are the dimensions of the texture, in pixels, minus one.

TXTLODBASE[0-(n-1)] must be set, where n is the number of LODs present. These registers specify the starting locations of the texture LODs in TRAM. They will usually be set during the texture load.

TXTTABCNTL must be set up properly. If texturing is turned off, then there will be only two inputs to the texture blender (PRIMCOLOR and PRIMALPHA) so any blends used can only have them as inputs. Typically, you will just want to set COLOROUT to PRIMCOLOR and ALPHAOUT to PRIMALPHA. If texturing is

turned on, then there are many many more options. However, if you actually want your texture to show up, you'll need to set COLOROUT to either TEXCOLOR or some BLEND one of whose inputs is TEXCOLOR.

ESCNTL must be used to turn perspective correction either on or off, according to your needs. If it is set incorrectly, your textures will be warped and unviewable.

Perspective

Turn perspective correction on:

```
CLT_SetRegister(pp, ESCNTL, CLA_ESCNTL (1,0,0))
```

Turn perspective correction off:

```
CLT_ClearRegister(pp, ESCNTL, CLA_ESCNTL (1,0,0))
```

Remember that if perspective correction is on, the texture coordinates entered in vertex commands must be multiplied by 1/w, but if perspective correction is off, raw texture coordinates must be used.

Sprites

To draw sprites, that is, to use the TE to simply draw textures onto the screen very quickly, make sure that blending is off, secondary source is off, perspective correction is off, Z-buffering is off (unless it's needed), and point filtering is being used. Be aware, however, that if you just want to copy large rectangular regions of data with the TE, it's faster to set a secondary source to point to the source data, enable source blending in the destination blender, and use a strip or fan of two untextured triangles to copy the data. However, this will only work if the source data is in literal 16- or 32-bit format, and if no special effects (such as black regions being transparent) are needed.

Lighting

To do lighting, you'll first need a 3D lighting model, which is a very complicated issue in its own right. However, there are a few typical things that might be done with the CLT to implement the graphical side of the lighting model. In particular, a lighting model will frequently be used with textured triangles, and it will want to do two things, namely darken the texture where the texture is not illuminated, and brighten the texture where there should be specular highlights.

To do either of these things on a triangle-by-triangle basis requires that there be color data for the vertices, but that color data can be used several different ways.

Lighting method #1: With this method, the RGB values of the color data are used to select the color of a light, and the alpha value sets how strong that light is. More specifically, the texture application blender is used, with the iterated alpha controlling the LERP between the iterated color and the textured color. So, to darken the texture, you can specify an RGB color of black. Then an alpha of 1 would result in black, an alpha of 0.5 would result in the texture being displayed at 50% intensity, an alpha of 0 would result in the normal texture, and so on. To brighten the texture, you can specify an RGB of white, or whatever color you want your specular highlights to be. The advantage of this method is that it uses only the texture application blender, so the destination blender is free to produce other effects, or to be turned off to save performance. Furthermore, this method allows you to still use your textured alpha for transparency or whatever, even though your iterated alpha is being taken up in the lighting. The disadvantage is that it assumes

that you've combined all of your lighting calculations and arrived at one final output color, instead of keeping the darkening and brightening effects separate. To use this method, simply set up the texture application blending as follows:

```
CLT_TXTTABCNTL(ptr, RC_TXTTABCNTL_FIRSTCOLOR_TEXCOLOR,
RC_TXTTABCNTL_SECONDCOLOR_PRIMCOLOR,
RC_TXTTABCNTL_THIRDCOLOR_PRIMALPHA, 0, 0,
RC_TXTTABCNTL_COLOROUT_BLEND,
RC_TXTTABCNTL_ALPHAOUT_TEXALPHA,
RC_TXTTABCNTL_BLENDOP_LERP);
```

Lighting method #2: With this method, the RGB color is used to darken the texture, and the alpha value is used to brighten the color. In this case, both the texture and destination blenders are used, with the darkening taking place in the texture blender and the lightening taking place in the destination blender. With this method, the RGB values are used as multipliers for the texture colors, so RGB values of (0.5, 0.5, 0.5) will result in the texture being displayed at half intensity. However, the alpha value is used to brighten the color, after the darkening has taken place, with an alpha of 0 not affecting the texture at all, and an alpha of 1 brightening all the way to solid white. This method is easier to use with most lighting models that treat specular highlights separately from the other lighting, but it uses both the texture and destination blenders, and doesn't allow the texture alpha to be accessed at all. To use this method, set up the blending as follows:

```
CLT_TXTTABCNTL(ptr, RC_TXTTABCNTL_FIRSTCOLOR_TEXCOLOR,
RC_TXTTABCNTL_SECONDCOLOR_PRIMCOLOR,
RC_TXTTABCNTL_THIRDCOLOR_PRIMALPHA, 0, 0,
RC_TXTTABCNTL_COLOROUT_BLEND,
RC_TXTTABCNTL_ALPHAOUT_PRIMALPHA,
RC_TXTTABCNTL_BLENDOP_MULT);

CLT_DBAMULTCNTL(ptr,
RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,
RC_DBAMULTCNTL_AMULTCOEFSELECT_CONST, 0, 0);

CLT_DBAMULTCONSTSSB0(GS_Ptr(gs), 255, 255, 255);
CLT_DBAMULTCONSTSSB1(GS_Ptr(gs), 255, 255, 255);

CLT_DBBMULTCNTL(ptr,
RC_DBBMULTCNTL_BINPUTSELECT_TEXCOLORCOMPLEMENT,
RC_DBBMULTCNTL_BMULTCOEFSELECT_TEXALPHA, 0, 0);

CLT_DBALUCNTL(ptr, RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0)
```

Simpler lighting models: If your lighting model doesn't require specular highlights, it is easy to simply have the RGB color or the alpha value multiply the texture value. This is accomplished with the texture application blender. For instance, to have the alpha value multiply the texture value, call:

```
CLT_TXTTABCNTL(ptr, RC_TXTTABCNTL_FIRSTCOLOR_TEXCOLOR,
RC_TXTTABCNTL_SECONDCOLOR_PRIMALPHA, 0, 0, 0,
RC_TXTTABCNTL_COLOROUT_BLEND,
RC_TXTTABCNTL_ALPHAOUT_TEXALPHA,
RC_TXTTABCNTL_BLENDOP_MULT);
```

Tiling a Texture

First of all, you can only tile a texture if its width or height (depending on which direction you're tiling in) is a power of two. To turn on tiling, you'll need to set the `TEXTUVMASK` and `TEXTUVMAX` registers in a fairly trick fashion. Each of these registers has both an X- and a Y-component. To tile your texture in the X direction, first set the X component of `TEXTUVMASK` to `txwidthmax-1`, where `txwidthmax` is equal to the width, in pixels, of the largest LOD. Then set the X component of `TEXTUVMAX` to `0x3ff >> (numLOD-1)`, where `numLOD` is the number of LODs present in the texture. To turn off tiling in the X direction, set the X component of `TEXTUVMASK` to `0x3ff` and the X component of `TEXTUVMAX` to `txwidth-1`, where `txwidth` is the width, in pixels, of the smallest (coarsest) LOD.

Similarly, to tile your texture in the Y direction, set the Y component of `TEXTUVMASK` to `txheightmax-1` and the Y component of `TEXTUVMAX` to `0x3ff >> (numLOD-1)`. To turn tiling off in the Y direction, set the Y component of `TEXTUVMASK` to `0x3ff` and the Y component of `TEXTUVMAX` to `txheight-1`.

Load an Uncompressed Texture

To do a texture load of an uncompressed texture, make the following calls:

```
CLT_TXTLDCNTL(ptr, 0, RC_TXTLDCNTL_LOADMODE_TEXTURE, bitoffset)
```

Where `bitoffset` is the offset into the byte in main memory of the start of the texture to be loaded, in bits.

```
CLT_TXTCOUNT(ptr, textureheight)
```

Where `textureheight` is the number of rows in the texture.

```
CLT_TXTLDSRCADDR(ptr, txaddress)
```

Where `txaddress` is the address in memory of the texture to be loaded.

```
CLT_TXTLODBASE0(ptr, tramaddress)
```

Where `tramaddress` is the word-aligned address in TRAM where the texture will be placed (ranging from 0 to 0x3fff). If you're only going to have one texture in TRAM at a time, this should probably be 0.

```
CLT_TXTLDWIDTH(ptr, txwidth, txsourcewidth)
```

Where `txwidth` is the width of the texture, in bits, and `txsourcewidth` is the width of the region from which the texture is being loaded, in bits.

```
CLT_TxLoad(ptr)
```

Load the texture.

Loading a PIP Table

To perform a PIP load, make the following calls:

```
CLT_TXTLODBASE0(ptr, 0x46000)
```

Tells the texture loader to load the PIP into PIP RAM.

```
CLT_TXTLDCNTL(ptr, 0, RC_TXTLDCNTL_LOADMODE_PIP, 0)
```

```
CLT_TXTLDSRCADDR(ptr, pipsourceaddress)
```

Where `pipsourceaddress` is the word-aligned address in main memory where the PIP is located.

```
CLT_TXTCOUNT(ptr, pipsize)
```

Where `pipsize` is the size, in bytes, of the PIP to load. A full 256 entry PIP takes up 1024 bytes.

```
CLT_TxLoad(ptr)
```

Sample Code

This is a sample program that uses Gstate and the Command List Toolkit. It draws random triangles up on the screen very rapidly.

```
/*All of the following are nice basic include files*/
#include <stdio.h>
#include <stdlib.h>
#include <kernel:types.h>
#include <graphics:clt:clt.h>
#include <graphics:clt:gstate.h>
#include <graphics:graphics.h>
#include <graphics:view.h>
#include <graphics:error.h>
#include <kernel:io.h>
#include <kernel:mem.h>
#include <kernel:random.h>
#include <assert.h>

#define TRISPERLIST 100

/*This constant specifies how many triangles each command list will draw. If
 *we wanted to we could have each command list we send to the TE have only
 *one triangle in it, but there is some overhead to sending lists, so it's
 *best to have each list not be tiny.
 */

int32 main(void)
{
    GState *gs;
    /*the heart of using GState is creating a GState struct... */

    Item viewItem; /*we need a View so that things will show up on screen*/
```

```
int i;

Item bitmaps[1];
/*we need a bitmap item for the view to display,
 *and for the TE to render into
 */

OpenGraphicsFolio (); /*must always be called before doing graphics...*/

gs=GS_Create();
/*This call creates and initializes the GState*/

GS_AllocLists(gs, 2, 4096);
/*This call allocates the command lists for the GState. In this case,
 *we'll have two command lists, each of 4096 words
 */

GS_AllocBitmaps( bitmaps, 320, 240, BMTYPE_16, 1, 0);
/*this call allocates a bitmap for us*/

GS_SetDestBuffer(gs, bitmaps[0]);
/*this call tells the GState which bitmap to render into*/

viewItem = CreateItemVA(MKNODEID(NST_GRAPHICS, GFX_VIEW_NODE),
    VIEWTAG_VIEWTYPE, BMTYPE_16,
    /*specifies 320x240x16bit*/

    VIEWTAG_BITMAP, bitmaps[0],
    /*Point the view towards our first bitmap*/

    TAG_END );
/*this call creates the View Item, which is necessary for
 *things to show up on screen
 */

AddViewToViewList( viewItem, 0 );
/*This causes the view to actually be displayed*/

/*the next several commands are all CLT commands, all of
 *which are register commands. For instance, the first
 *command is CLT_DBUSERCONTROL. It's passed a pointer to
 *a command list, and a bunch of numbers. What it does is
 *it adds a command to the command list. That command
 *sets the state of the DBUSERCONTROL register, which is
 *nice, basic TE register. The value it sets DBUSERCONTROL
 *to comes from the arguments. In the case of DBUSERCONTROL,
 *most of these arguments represent a single flag bit.
 */

/*note that the moral of this story is that before _anything_
 *will show up on the screen, there are quite a few registers
 *that need to be set up, but once they're all set up, they
 *can be more or less ignored
 */
```

```

CLT_DBUSERCONTROL(GS_Ptr(gs), 0,0,0,0,1,1,0,15);
/*basic initialization. In this case, Z-buffering is off,
 *window clipping is off, destination blending is on, source
 *blending is on, and dithering is off
 */

/*this next command also is a register setting command,
 *in this case setting the DBDISCARDCONTROL register
 */
CLT_DBDISCARDCONTROL(GS_Ptr(gs),0,0,0,0);
/*all pixel discards are turned off. these would be turned on, for
 *instance, to make all black pixels transparent
 */

CLT_DBCONSTIN(GS_Ptr(gs), 0,0,0);
/*set the constant color used in destination blending to black
 *(destination blending takes the color and texture from the triangle
 *and does a number of odd things to it. In this case, we'll just add
 *it to black, which effectively leaves it unchanged)
 */

CLT_DBAMULTCONSTSSB0(GS_Ptr(gs), 0xff, 0xff, 0xff);
CLT_DBAMULTCONSTSSB1(GS_Ptr(gs), 0xff, 0xff, 0xff);
CLT_DBBMULTCONSTSSB0(GS_Ptr(gs), 0xff, 0xff, 0xff);
CLT_DBBMULTCONSTSSB1(GS_Ptr(gs), 0xff, 0xff, 0xff);
/*set all of our multiplying constants for the destination blender
 *to 1, so we can just pass colors directly through
 */

CLT_DBALUCNTL(GS_Ptr(gs), RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0);

/*This set the final ALU stage of the destination blending to just add the
 *A and B colors together. In this case, the A color will be the
 *shading from the triangle we're drawing and the B color will be black
 */

CLT_TXTADDRCNTL(GS_Ptr(gs), 0, 0, 0, 0, 0);
/*Turn texturing off*/

CLT_DBAMULTCNTL(GS_Ptr(gs), RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,
RC_DBAMULTCNTL_AMULTCOEFSELECT_CONST,
RC_DBAMULTCNTL_AMULTCONSTCONTROL_TEXSSB, 0);

/*sets the A input to the destination blender to be the color from the texture
 *mapper. Note that even though we aren't doing any texturing we want the color
 *from the texture mapper, because the texture mapper handles all shading.
 */

CLT_DBBMULTCNTL(GS_Ptr(gs), RC_DBBMULTCNTL_BINPUTSELECT_CONSTCOLOR,
RC_DBBMULTCNTL_BMULTCOEFSELECT_CONST,
RC_DBBMULTCNTL_BMULTCONSTCONTROL_TEXSSB, 0);

/*sets the B input to the destination blender to just be a constant color

```

```
    *(which in this case we earlier set to black)
    */

    GS_SendList(gs);

/*This command is totally different from all the CLT macros we've been using.
*It tells the GState to take the command list it's been assembling, which
*consists of a bunch of register instructions, and send it to the TE.
*Note that whenever you call SendList the Gstate will stick a CLT_Pause
*command at the end of your command list, which means that you don't have
*to remember to do so yourself. But if you're ever sending lists
*yourself, always remember to terminate them with a CLT_Pause command.
*/

    while(1)
    {
        for (i=0;i<TRISPERLIST;i++)
        {

            CLT_TRIANGLE(GS_Ptr(gs), 1, RC_STRIP, 0, 0, 1, 3);
            /*To set up a command list to draw a triangle, we first need a
            *triangle command. The second argument says that this is a new
            *tristrip/fan (as opposed to connecting to a previously drawn one.
            *The third argument says that it's a strip, not a fan.
            *The fourth argument says that it has no perspective info (z
            *values). The fifth argument says that it has no texturing.
            *The sixth argument says that it does have RGBA color info.
            *The final argument says that it has 3 vertices.
            *
            *After the triangle command we'll need to issue one vertex command
            * for each vertex of the triangle. Note that there can not be any
            *other TE instructions between the triangle command and the vertex
            *commands. Also, if you give the wrong number of vertex commands,
            *or if you, say, include
            *perspective info in the vertex command but don't mention it in the
            *triangle command, then bad things will happen. Namely, the TE will
            *probably time out.
            */

            CLT_VertexRgba(GS_Ptr(gs), (gfloat)(rand()%320), (gfloat)(rand()%240),
                (gfloat)(rand() % 10000)/10000.0,
                (gfloat)(rand() % 10000)/10000.0,(gfloat)(rand() % 10000)/10000.0,0);
            CLT_VertexRgba(GS_Ptr(gs), (gfloat)(rand()%320), (gfloat)(rand()%240),
                (gfloat)(rand() % 10000)/10000.0,
                (gfloat)(rand() % 10000)/10000.0,(gfloat)(rand() % 10000)/10000.0,0);
            CLT_VertexRgba(GS_Ptr(gs), (gfloat)(rand()%320), (gfloat)(rand()%240),
                (gfloat)(rand() % 10000)/10000.0,
                (gfloat)(rand() % 10000)/10000.0,(gfloat)(rand() % 10000)/
10000.0,0);

            /*CLT_VertexRgba is the macro for vertices with shading but no perspective
            *or texture. Since perspective, texture, and shading can all be turned
            *on or off independently, there are a total of 8 possible distinct
            *types of vertex command, each with an accompanying macro.
            */
        }
    }
```

```
    CLT_Sync(GS_Ptr(gs));

    /*after drawing a triangle, but before setting any registers, you should
    *always call CLT_Sync, which inserts a sync command into the command list.
    *In this particular case, we don't set any registers after drawing the
    *triangle, so we don't really need this sync command, but I needed an
    *excuse to mention it, since it's so important. Get in the habit of
    *sticking it after triangle commands.
    */
    }

    GS_SendList(gs);
    /*This sends our newly created command list with instructions for
    * drawing 100 triangles to the Triangle Engine
    */

    }
    CloseGraphicsFolio();
    /*and we're done!*/

    }
    /*have a nice day*/
```


Function Calls

This Appendix lists all of the Command List Toolkit function calls.

GState

GS_AllocBitmaps	Utility to allocate frame buffers and Z-buffer
GS_AllocLists	Allocate one or more command list buffers for graphics rendering
GS_BeginFrame	Notifies a GState that the next cmd. list is the first for a frame
GS_Create	Creates a GState (Graphics State) object
GS_Delete	Frees all memory associated with a GState object
GS_FreeBitmaps	Utility to free bitmaps and Z-buffer
GS_FreeLists	Free the memory used by command lists for a GState object
GS_GetCmdListIndex	Returns index of which cmd list buffer is in use
GS_GetCount	Get the current GState counter
GS_GetCurListStart	Get ptr to start of the current command list buffer
GS_GetDestBuffer	Get the Item number of the bitmap being used as an output frame buffer
GS_GetVidSignal	Returns the signal bit which a GState is using for video synchronization
GS_GetZBuffer	Get the Item number of the bitmap being used as a Z-buffer
GS_Init OBSOLETE	Initializes a GState object to a default state
GS_Reserve	Reserve block of memory in GState's current command list buffer
GS_SendList	Send a GState's current command list to the Triangle Engine
GS_SetDestBuffer	Set a bitmap as the output frame buffer for a GState
GS_SetList	Set a GState to use a new command list buffer
GS_SetVidSignal	Attach a signal to a GState object, for video synchronization
GS_SetZBuffer	Set a bitmap as the Z-buffer for a GState
GS_WaitIO	Wait for all pending IO to complete for a GState

Triangle Engine Command List Toolkit

CLT_AllocSnippet Allocate space for the data for a CltSnippet
 CLT_ClearFrameBuffer Clear frame buffer and/or Z-buffer via CLT cmds
 CLT_ComputePipLoadCmdListSize Compute size of PIP load command list
 CLT_ComputeTxLoadCmdListSize Compute size of texture load command list
 CLT_CopySnippetData Copy data portion of a cmd list snippet to specified location
 CLT_CreatePipCommandList Creates a cmd list snippet to load a PIP
 CLT_CreateTxLoadCommandList Creates a cmd list snippet to load a texture
 CLT_FreeSnippet Free memory allocated for a CltSnippet's data
 CLT_InitSnippet Initializes a CltSnippet structure
 CLT_SetSrcToCurrentDest Set the Dest Blender so that src blends will occur with cur frame buffer

Globals

CltEnableTextureSnippet Global var. used to enable texturing
 CltNoTextureSnippet Global var. used to disable texturing

GS_AllocBitmaps

Utility to allocate frame buffers and Z-buffer

Synopsis

```
Err GS_AllocBitmaps(Item bitmaps[], uint32 xres, uint32 yres,  
    uint32 bmType, uint32 numFrameBufs, bool useZb);
```

Description

Utility routine to allocate frame buffers, and, optionally, also a Z-buffer. Since the Triangle Engine is able to perform better when the Z-buffer is aligned to an odd page offset from the frame buffer(s), **GS_AllocBitmaps()** will handle aligning the buffers automatically.

Arguments

bitmaps

Array of Items which will become the bitmaps. This array should be large enough to contain one item per frame buffer, plus one more if Z-buffering is going to be used. Upon successful return, bitmaps will contain the specified number of frame buffers in bitmaps[0] ... bitmaps[numFrameBufs-1]. If a Z-buffer is needed, it will be allocated in bitmaps[numFrameBufs].

xres

The width, in pixels, of each bitmap

yres

The height, in pixels, of each bitmap

bmType

Specifies the type of bitmaps to be created. This value should be one of the constants defined in the <graphics:bitmap.h> header file, such as BMTYPE_32 or BMTYPE_16.

numFrameBufs

The number of frame buffers to be allocated. This number should NOT include the Z-buffer

useZb

Specifies whether a Z-buffer should be allocated. If TRUE, then the size of the bitmaps[] array should be numFrameBufs+1.

Return Value

Returns ≥ 0 for success or a negative error code for failure

Implementation

DLL call implemented in gstate V29

Associated Files

<graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_FreeBitmaps(), GS_SetDestBuffer(), GS_SetZBuffer()

GS_AllocLists

Allocate one or more command list buffers for graphics rendering

Synopsis

```
Err GS_AllocLists(GState* g, uint32 nLists, uint32 listSize);
```

Description

Allocates one or more command lists buffers of the specified size. These buffers are used by the Font Folio, 2D Graphics Folio, and 3D Graphics Library to build command lists for the Triangle Engine to render.

Each command list buffer must be large enough that the largest triangle strip or fan in a program's data can fit within the buffer. Triangle strips and fans can sometimes be several hundred vertices long, and each vertex can take up to 9 32-bit words (36 bytes) of command list buffer space.

Normally, a minimum of two command list buffers should be allocated, so that while the Triangle Engine is rendering one buffer's commands, the CPU can begin preparing the next set of rendering commands. Larger buffers will often yield better performance than smaller buffers, since the CPU spends less time flushing buffers and waiting for new ones to be freed up by the Triangle Engine. The parameters to **GS_AllocLists()** should be adjusted during development of a title, to optimally blend between high performance and memory usage.

Arguments

- `g`
Pointer to a GState object
- `nLists`
Number of command lists buffers to allocate.
- `listSize`
Size of each list to be allocated, in WORDS.

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

[<graphics:clt:gstate.h>](graphics:clt:gstate.h), System.m2/Modules/gstate

See Also

[GS_Create\(\)](#), [GS_FreeLists\(\)](#), [GS_Resume\(\)](#), [GS_SendList\(\)](#), [GS_SetList\(\)](#)



[GS_AllocBitmaps](#)



[GS_BeginFrame](#)

GS_BeginFrame

Notifies a GState that the next cmd. list is the first for a frame

Synopsis

Err **GS_BeginFrame**(GState* g);

Description

Notifies a GState that the next command list buffer is the first to be rendered to a Bitmap. This routine is used when double- buffering is desired. At the first call to **GS_SendList()** after **GS_BeginFrame()** is called, the GState will wait on a signal from the Display Folio before sending a command list to the Triangle Engine device driver.

In order for double-buffering to correctly prevent screen tearing, this routine should be called after each call to switch between frame buffers (usually with a call to **ModifyGraphicsItem()**). In addition, a signal should be associated with the View and a GState when these objects are being created. See **GS_SetVidSignal()** for more information about how to correctly allocate this signal.

Arguments

g
Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_SetVidSignal()**, **GS_GetVidSignal()**, **GS_SendList()**



GS AllocLists



GS Create

GS_Create

Creates a GState (Graphics State) object

Synopsis

```
GState* GS_Create(void);
```

Description

Creates a GState, or Graphics State, object. This object is used to encapsulate the information which needs to be maintained for command list buffers. The Triangle Engine operates by executing the series of instructions which are placed into these command list buffers.

The 2D graphics folio, font folio, and 3D graphics library all use command list buffers to communicate with the Triangle Engine. By creating a common GState that can be shared by these folios and libraries, the user can combine text, 2D graphics, and 3D graphics more easily.

After a GState is created, it still must be initialized by calling GS AllocLists(), to create one or more buffers of the specified size.

Return Value

Returns a pointer to a GState if successful, or NULL if it fails.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS AllocLists(), GS Reserve(), GS SendList(), GS SetList(), GS WaitIO()



GS_BeginFrame



GS_Delete

GS_Delete

Frees all memory associated with a GState object

Synopsis

```
Err GS_Delete(GState* g);
```

Description

Frees up all memory used by a GState object. If command list buffers have been allocated by calling [GS AllocLists\(\)](#), [GS Delete\(\)](#) will also free the memory used by these buffers.

Arguments

g
Pointer to a GState object

Return Value

Returns 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

[GS Create\(\)](#), [GS AllocLists\(\)](#), [GS FreeLists\(\)](#)



[GS Create](#)



[GS FreeBitmaps](#)

GS_FreeBitmaps

Utility to free bitmaps and Z-buffer

Synopsis

```
Err GS_FreeBitmaps(Item bitmaps[], uint32 numBitmaps);
```

Description

Free all bitmaps in the bitmaps[] array. Usually, these will be bitmaps that were created by calling **GS_AllocBitmaps()**.

Note that if any of these bitmaps were in use by a GState object, an error will be returned the next time that GState needs to use one of the bitmaps. Therefore, it is a good idea to always either assign new bitmaps to a GState before freeing its bitmaps, or to free a GState, by calling **GS_Delete()**, before freeing the array of bitmaps.

Arguments

bitmaps

Array of items representing the bitmaps for the frame buffer(s), and optionally, a Z-buffer.

numBitmaps

The total number of frame buffers in the bitmaps[] array. This number should equal the number of frame buffers allocated, plus one if a Z-buffer was used.

Return Value

Returns != 0 for success or a negative error code for failure

Implementation

DLL call implemented in gstate V29

Associated Files

graphics.clt:gstate.h, System.m2/Modules/gstate

See Also

GS_Create(), **GS_Delete()**, **GS_AllocBitmaps()**



GS_Delete



GS_FreeLists

GS_FreeLists

Free the memory used by command lists for a GState object

Synopsis

Err **GS_FreeLists**(GState* g);

Description

Free the memory used by the command list buffer(s) for a GState object. These buffers should have been allocated by calling **GS_AllocLists()**.

Normally, this routine does not need to be called, since it will be called by **GS_Delete()** if the command list buffers are still allocated. However, it can be used in special situations, such as when, because of memory considerations at run-time, it becomes necessary to re-size the command list buffers.

Arguments

g
Pointer to a GState object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_Delete()**



GS_FreeBitmaps



GS_GetCmdListIndex

GS_GetCmdListIndex

Returns index of which cmd list buffer is in use

Synopsis

```
uint32 GS_GetCmdListIndex(GState* g)
```

Description

This routine returns the index into the command list buffer array of the command list buffer which is currently in use. In most cases, this routine will only be useful to people writing their own routine to send a command list to the Triangle Engine.

In the case where more than one command list buffer is allocated when **GS AllocLists()** is called, each call to **GS SendList()** will increment this index so that the CPU can begin filling the next command list buffer.

Arguments

g
Pointer to a GState object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS Create(), **GS AllocLists()**, **GS SendList()**



GS_FreeLists



GS_GetCount

GS_GetCount

Get the current GState counter.

Synopsis

```
uint32 GS_GetCount(GState *gs);
```

Description

Returns the number of times that GS_WaitIO() has completed.

Arguments

g
Pointer to a GState object

Return Value

Returns the count value. If the gs parameter is invalid, then the return value will be garbage.

Implementation

DLL call implemented in gstate V30

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate



GS_GetCmdListIndex



GS_GetCurListStart

GS_GetCurListStart

Get ptr to start of the current command list buffer

Synopsis

CmdListP **GS_GetCurListStart**(GState* g)

Description

Returns a pointer to the start of the current command list buffer. In most cases, this routine will only be useful to people writing their own routine to send a command list to the Triangle Engine.

Arguments

g
Pointer to a GState object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_SendList()



GS_GetCount



GS_GetDestBuffer

GS_GetDestBuffer

Get the Item number of the bitmap being used as an output frame buffer

Synopsis

```
Item GS_GetDestBuffer(GState* g);
```

Description

Returns the Item number for the Bitmap being used as an output frame buffer for a GState object. When Triangle Engine commands are rendered through the specified GState object, the Triangle Engine will render into this Bitmap. Z-buffer information is stored in a separate bitmap.

Arguments

g
Pointer to a GState object

Return Value

Returns an Item number for the Bitmap, or an error code if a valid bitmap has not been set for this GState.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

[GS_Create\(\)](#), [GS_SetDestBuffer\(\)](#), [GS_SetZBuffer\(\)](#), [GS_GetZBuffer\(\)](#)



[GS_GetCurListStart](#)



[GS_GetVidSignal](#)

GS_GetVidSignal

Returns the signal bit which a GState is using for video synchronization

Synopsis

```
int32 GS_GetVidSignal(GState* g);
```

Description

Returns a signal bit mask showing which signal, if any, is being used by a GState object to keep Triangle Engine rendering in sync with double-buffering of bitmaps by the Display Manager. This signal is associated with a GState by calling **GS_SetVidSignal()**.

This signal should normally be the same value which was passed to the Display Folio as the arg for the VIEWTAG_DISPLAYSIGNAL tag, when the View was created.

Arguments

g
Pointer to a GState object

Return Value

Returns the signal associated with a GState, or 0 if one was not associated with the GState.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_SetVidSignal()**, **GS_BeginFrame()**, **GS_SendList()**



GS_GetDestBuffer



GS_GetZBuffer

GS_GetZBuffer

Get the Item number of the bitmap being used as a Z-buffer

Synopsis

```
Item GS_GetZBuffer(GState* g);
```

Description

Returns the Item number for the Bitmap being used as a Z-buffer for a GState object. When Triangle Engine commands are rendered through the specified GState object, the Triangle Engine will use this bitmap to hold Z-buffer information. The actual rendered scene is stored in a separate bitmap.

Arguments

g
Pointer to a GState object

Return Value

Returns an Item number for the Bitmap, or an error code if a valid bitmap has not been set for this GState.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_SetZBuffer(), GS_SetDestBuffer(), GS_GetDestBuffer()



GS_GetVidSignal



GS_Init

GS_Init

OBSOLETE - Initializes a GState object to a default state

Synopsis

```
Err GS_Init(GState* g);
```

Description

This call is OBSOLETE. Use [GS_Create\(\)](#) instead, to create a GState object.

Arguments

g Pointer to a GState object

Return Value

Returns 0 for success or a negative error code for failure.

Implementaion

No longer implemented.

Associated Files

[<:graphics:clt:gstate.h>](#), [System.m2/Modules/gstate](#)

See Also

[GS_Create\(\)](#)



[GS_GetZBuffer](#)



[GS_Reserve](#)

GS_Reserve

Reserve block of memory in GState's current command list buffer

Synopsis

```
void GS_Reserve(GState* g, uint32 nwords);
```

Description

Check to ensure that there is enough space in a GState's current command list buffer for the requested number of words. If there isn't enough space, flush the current command list buffer, and if more than one command list buffer was allocated by [GS AllocLists\(\)](#), reserve the requested space in the next available buffer.

Note that this routine will silently fail if the requested number of words is larger than the total size of a command list buffer.

Arguments

g
Pointer to a GState object

nwords
Number of words to reserve

Implementation

DLL call implemented in gstate V29

Associated Files

[<:graphics:clt:gstate.h>](#), [System.m2/Modules/gstate](#)

See Also

[GS Create\(\)](#), [GS AllocLists\(\)](#), [GS SendList\(\)](#), [GS SetList\(\)](#)



[GS Init](#)



[GS SendList](#)

GS_SendList

Send a GState's current command list to the Triangle Engine

Synopsis

```
Err GS_SendList(GState* g);
```

Description

Send the data in the GState's current command list buffer to the Triangle Engine device driver to be rendered. If more than one command list buffer was allocated when **GS AllocLists()** was called, the GState will use the next available command list buffer when this function returns.

The Font Folio, 2D Graphics Folio, and 3D Graphics Library build buffers of Triangle Engine instructions using CLT, or the Command List Toolkit. Once the buffer fills up, or gets close to becoming full, **GS_SendList()** is called to flush the buffer and let the Triangle Engine begin rendering.

To support tear-free double-buffering of the frame buffers, **GS_SendList()** will wait on a signal from the video hardware, if one was associated with a GState by calling **GS_SetVidSignal()**. This wait will only occur when the first command list buffer is to be rendered into a frame. To specify that this wait should occur, the user should call **GS_BeginFrame()** whenever a new frame buffer has just been displayed.

Normally, the user application should call `g->gs_SendList(g)` instead of calling this routine directly, so that if it becomes necessary to debug the command list output, a user function can easily replace **GS_SendList()**. `g->gs_SendList()` is initialized to use **GS_SendList(g->gs_SendList())** is initialized to use **GS_SendList()** by default when a GState is created.

`g->gs_SendList()` is called automatically by the GState folio whenever `g->gs_SendList()` is called automatically by the GState folio whenever **GS_Resume()** cannot allocate enough space in the current command list buffer.

Arguments

`g`
Pointer to a GState object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in `gstate V29`

Associated Files

`<:graphics:clt:gstate.h>`, `System.m2/Modules/gstate`

See Also

GS_Create(), **GS_AllocLists()**, **GS_Resume()**, **GS_SetList()**, **GS_SetVidSignal()**, **GS_GetVidSignal()**, **GS_BeginFrame()**

GS_SetDestBuffer

Set a bitmap as the output frame buffer for a GState

Synopsis

```
Err GS_SetDestBuffer(GState* g, bmItem);
```

Description

Set a bitmap as the output frame buffer for a GState. When commands are sent to the Triangle Engine by this GState object, the output from the Triangle Engine will go into the specified bitmap.

Arguments

g
Pointer to a GState object

bmItem
Item representing a Bitmap object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

[<:graphics:clt:gstate.h>](clt:gstate.h), System.m2/Modules/gstate

See Also

[GS_Create\(\)](#), [GS_GetDestBuffer\(\)](#), [GS_SetZBuffer\(\)](#), [GS_GetZBuffer\(\)](#)



[GS_SendList](#)



[GS_SetList](#)

GS_SetList

Set a GState to use a new command list buffer

Synopsis

```
Err GS_SetList(GState* g, uint32 idx);
```

Description

Set a GState to use a new command list buffer. These buffers should be allocated with [GS_AllocLists\(\)](#). [GS_SendList\(\)](#) will call this routine automatically to set up the GState to use the next available command list buffer, if more than one buffer has been allocated.

Arguments

g
Pointer to a GState object

idx
Index stating which command list buffer should be used. Value should be from 0 ... (numberCmdLists - 1)

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

[<:graphics:clt:gstate.h>](#), [System.m2/Modules/gstate](#)

See Also

[GS_Create\(\)](#), [GS_AllocLists\(\)](#), [GS_Resume\(\)](#), [GS_SendList\(\)](#)



[GS_SetDestBuffer](#)



[GS_SetVidSignal](#)

GS_SetVidSignal

Attach a signal to a GState object, for video synchronization

Synopsis

```
Err GS_SetVidSignal(GState* g, int32 signal);
```

Description

Associate a signal bit with a GState object. This signal will allow a GState to stay synchronized with the video in a double- buffering scheme.

To correctly enable this feature of GState, allocate a signal by calling [AllocSignal\(\)](#) before a View is created. Then, use this allocated signal as the argument for the VIEWTAG_DISPLAYSIGNAL tag when a view is created. Last, associate this signal with a GState by calling [GS_SetVidSignal\(\)](#). Whenever one bitmap has been fully rendered and sent to the Display Folio, the user should also call [GS_BeginFrame\(\)](#), to tell a GState that it needs to wait for this signal before it sends the next command list.

Arguments

g
Pointer to a GState object

signal
A signal, created by [AllocSignal\(\)](#)

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

[<:graphics:clt:gstate.h>](#), System.m2/Modules/gstate

See Also

[GS_Create\(\)](#), [GS_GetVidSignal\(\)](#), [GS_BeginFrame\(\)](#), [GS_SendList\(\)](#)



[GS_SetList](#)



[GS_SetZBuffer](#)

GS_SetZBuffer

Set a bitmap as the Z-buffer for a GState

Synopsis

```
Err GS_SetZBuffer(GState* g, Item bmlItem);
```

Description

Set a bitmap as the Z-buffer for a GState. If Z-buffering is enabled for the Triangle Engine, this bitmap will be used to hold the depth information for triangles as they are rendered.

Arguments

g
Pointer to a GState object

bmlItem
Item representing a Bitmap object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_GetZBuffer(), GS_SetDestBuffer(), GS_GetDestBuffer()



GS_SetVidSignal



GS_WaitIO

GS_WaitIO

Wait for all pending IO to complete for a GState

Synopsis

Err GS_WaitIO(GState* g);

Description

For a given GState, wait for any active IORequests to complete. This routine is usually used to ensure that the Triangle Engine is not still rendering into a bitmap before that bitmap is displayed.

Arguments

g
Pointer to a GState object

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_AllocLists(), GS_SendList()



GS_SetZBuffer

CLT_AllocSnippet

Allocate space for the data for a CltSnippet

Synopsis

```
Err CLT_AllocSnippet(CltSnippet* s, uint32 nWords);
```

Description

Allocate space for the specified number of words of command list data for a given CltSnippet. The CltSnippet should already have been initialized by calling [CLT_InitSnippet\(\)](#).

Arguments

s
Pointer to a valid, initialized CltSnippet

nWords
Number of words of memory to be allocated for the CltSnippet

Return Value

Returns 0 for success or a negative error code for failure.



Caveats

If the CltSnippet already has memory allocated for its data, this memory will be left dangling in memory. Use [CLT_FreeSnippet\(\)](#) to free the memory first.

Implementation

Library call in libclt.a

See Also

[CLT_FreeSnippet\(\)](#), [CLT_InitSnippet\(\)](#)



[CLT_ClearFrameBuffer](#)

CLT_ClearFrameBuffer

Clear frame buffer and/or Z-buffer via CLT cmds

Synopsis

```
void CLT_ClearFrameBuffer(GState* gs, float red, float green,  
    float blue, float alpha, bool clearScreen, bool clearZ);
```

Description

Clear the frame buffer and/or Z-buffer via the Triangle Engine. This is done by drawing two large triangles of the specified color. If clearing of the Z-buffer is desired, 0 is written to the whole Z-buffer as well. Note that because the screen clear is done via Triangle Engine commands, the clear doesn't actually occur until the command list is sent to the Triangle Engine.

Arguments

- gs**
Pointer to a GState object
- red, green, blue, alpha**
Floating point numbers between 0.0 and 1.0, inclusive, which specify the intensities of the R, G, B, and alpha channels, respectively.
- clearScreen**
Boolean specifying whether the screen should be cleared. This would be FALSE when the user only wants to clear the Z-buffer.
- clearZ**
Boolean specifying whether the Z-buffer should be cleared. Note that this value is ignored when a Z-buffer is not being used. Also, when a Z-buffer is being used, this routine will NOT enable Z-buffering if clearZ is set to TRUE. It is the responsibility of the user's code to enable Z-buffering before calling this routine. This is done because normally, the Z-buffer would just be enabled once when the application first starts up, whereas clearing the Z-buffer would be done once per frame.

Implementation

Library call in libclt.a

See Also

[GS_Create\(\)](#), [GS_SendList\(\)](#)



[CLT_AllocSnippet](#)



[CLT_ComputePipLoadCmdListSize](#)

CLT_ComputePipLoadCmdListSize

Compute size of PIP load command list

Synopsis

```
int32 CLT_ComputePipLoadCmdListSize(CltPipControlBlock* txPipCB);
```

Description

Computes the size of a PIP load command list snippet, without actually creating the snippet. This routine is most useful when users want to have a PIP load command list be created directly into the command list buffer, and need to know how much space will be needed ahead of time to prevent overflowing the command list buffer.

Arguments

txPipCB

Pointer to a CltPipControlBlock structure. See [CLT_CreatePipCommandList\(\)](#) for information on how to correctly fill in this structure.

Return Value

Returns the necessary size, in words, if successful, or a negative error code if the call fails.

Implementation

Library call in libclt.a

See Also

[CLT_CreatePipCommandList\(\)](#), [GS_Reserve\(\)](#)



[CLT_ClearFrameBuffer](#)



[CLT_ComputeTxLoadCmdListSize](#)

CLT_ComputeTxLoadCmdListSize

Compute size of texture load command list

Synopsis

```
int32 CLT_ComputeTxLoadCmdListSize(CltTxLoadControlBlock* txLoadCB);
```

Description

Computes the size of a texture load command list snippet, without actually creating the snippet. This routine is most useful when users want to have a texture load command list be created directly into the command list buffer, and need to know how much space will be needed ahead of time to prevent overflowing the command list buffer.

Arguments

txLoadCB

Pointer to a CltTxLoadControlBlock structure. See [CLT_CreateTxLoadCommandList\(\)](#) for information on how to correctly fill in this structure.

Return Value

Returns the necessary size, in words, if successful, or a negative error code if the call fails.

Implementation

Library call in libclt.a

See Also

[CLT_CreateTxLoadCommandList\(\)](#), [GS_Reserve\(\)](#)



[CLT_ComputePipLoadCmdListSize](#)



[CLT_CopySnippetData](#)

CLT_CopySnippetData

Copy data portion of a cmd list snippet to specified location

Synopsis

```
void CLT_CopySnippetData(uint32** dest, CltSnippet* src)
```

Description

Copies data from a CltSnippet to another memory location, updating the dest ptr as it goes. Most often, this routine is used to copy command list snippets into a command list buffer within the GState.

Arguments

dest

Pointer to a pointer to a buffer of 32-bit quantities. On exit from this routine, *dest will be updated to point to just beyond the copied data area, so that a subsequent call to **CLT_CopySnippetData()** can be made without updating the pointer manually.

src

Pointer to a valid, initialized CltSnippet

Implementation

Library call in libclt.a

See Also

GS_Reserve(), **GS_SendList()**, **CLT_InitSnippet()**



CLT_ComputeTxLoadCmdListSize



CLT_CreatePipCommandList

CLT_CreatePipCommandList

Creates a cmd list snippet to load a PIP

Synopsis

```
Err CLT_CreatePipCommandList(CltPipControlBlock* txPipCB);
```

Description

Creates a command list snippet to load the specified block of memory as a PIP, or Pen Index Palette, for indexed textures to use. The fields of the CltPipControlBlock should be filled in as follows:

pipData
 Pointer to the PIP data in RAM

pipIndex
 Starting index of the data to load

pipNumEntries
 Number of entries in the data to be loaded

Arguments

txPipCB
 Pointer to a CltPipControlBlock structure. On entry, txPipCB should be filled in as described above. On exit, the field pipCommandList will be filled in with a command list snippet that can be copied into a command list buffer when the PIP load needs to occur. See [CLT_CopySnippetData\(\)](#) for more details.

Return Value

Returns 0 for success or a negative error code for failure.

Implementation

Library routine in libclt.a

See Also

[CLT_CopySnippetData\(\)](#), [CLT_ComputePipLoadCmdListSize\(\)](#)



[CLT_CopySnippetData](#)



[CLT_CreateTxLoadCommandList](#)

CLT_CreateTxLoadCommandList

Creates a cmd list snippet to load a texture

Synopsis

```
Err CLT_CreateTxLoadCommandList(CltTxLoadControlBlock* txLoadCB);
```

Description

Creates a command list snippet to load the specified texture into the Triangle Engine's TRAM, or Texture RAM. For uncompressed textures which do not require runtime carving, the data for the one or more levels of detail is transferred as one large block of contiguous RAM, for optimal performance. For indexed textures, the PIP must be loaded separately, by calling [CLT_CreatePipCommandList\(\)](#). The fields of the txLoadCB field should be filled in as follows:

textureBlock	Pointer to a CltTxData block. This block contains data about the actual texels, such as num LODs, texture format, etc.
numLOD	Number of LODs (Levels of Detail) in texture
XWrap	Wrap mode in X-direction (0=Clamp, 1=Tile)
YWrap	Wrap mode in Y-direction (0=Clamp, 1=Tile)
XSize	Width of area to load. Normally equal to the minX field of the textureBlock structure
YSize	Height of area to load. Normally equal to the minY field of the textureBlock structure
XOffset	Left edge of sub-texture to load. For non-carved textures, this will normally be 0.
YOffset	Top edge of sub-texture to load. For non-carved textures, this will normally be 0.
tramOffset	Offset into the 16k TRAM where the texture should be placed. For uncompressed, uncarved textures, this value must be 32-bit aligned.
tramSize	Size texture will require in TRAM.

Arguments

txLoadCB	Pointer to a CltTxLoadControlBlock structure. On entry, txLoadCB must be filled in as described above. On exit, the field lcbCommandList will contain a command list snippet to load and use the specified texture. Additionally, the useCommandList can be used to just re-use the texture, assuming that it is already loaded. The useCommandList is most useful when multiple textures reside in the TRAM at once, and the user data needs to alternate between these different textures.
----------	--

Return Value

Returns 0 for success or a negative error code for failure.

Implementation

Library routine in libclt.a

See Also

[CLT_CopySnippetData\(\)](#), [CLT_ComputeTxLoadCmdListSize\(\)](#), [CLT_CreatePipCommandList\(\)](#)



[CLT_CreatePipCommandList](#)



[CLT_FreeSnippet](#)

CLT_FreeSnippet

Free memory allocated for a CltSnippet's data

Synopsis

```
void CLT_FreeSnippet(CltSnippet* s);
```

Description

Frees the memory allocated to a CltSnippet's data area. The memory is only freed if it was allocated for the snippet. That is, there are times when several snippets may all share the same data area, such as when one is only supposed to represent a smaller portion of another. In that case, calling **CLT_FreeSnippet()** on the smaller sub-snippet will not cause the smaller portion of memory to be freed from the larger memory block.

Arguments

s Pointer to a valid, initialized CltSnippet



Caveats

In the case where a snippet is using memory from a larger snippet, and **CLT_FreeSnippet()** is called to free the memory used by the larger snippet, there is no way to mark the smaller snippet that is sharing the memory as freed. This could result in a CltSnippet pointing to garbage memory.

Implementation

Library call implemented in libclt.a

See Also

CLT_AllocSnippet()



CLT_CreateTxLoadCommandList



CLT_InitSnippet

CLT_InitSnippet

Initializes a CltSnippet structure

Synopsis

```
void CLT_InitSnippet(CltSnippet* s);
```

Description

Initializes a CltSnippet structure. The structure must be created before calling this routine, either by calling **AllocMem()** or by just declaring a variable of type CltSnippet. Once initialized, a CltSnippet can be used in many other CLT calls.

Note that initializing a CltSnippet does not allocate any memory for the snippet's data space. To allocate that memory, call **CLT AllocSnippet()** after calling **CLT_InitSnippet()**.

Arguments

s
Pointer to a valid CltSnippet

Implementation

Library call in libclt.a

See Also

CLT AllocSnippet(), **CLT_CopySnippet()**, **CLT_CopySnippetData()**, **CLT_FreeSnippet()**, **AllocMem()**



CLT_FreeSnippet



CLT_SetSrcToCurrentDest

CLT_SetSrcToCurrentDest

Set the Dest Blender so that src blends will occur with cur frame buffer

Synopsis

```
void CLT_SetSrcToCurrentDest(GState* g);
```

Description

This routine adds some commands to the current command list buffer of a GState which set the destination blend source buffer equal to the current destination frame buffer. This is usually most useful when trying to make objects appear translucent. To achieve a translucent effect, an object gets blended with the current frame buffer in the destination blender of the Triangle Engine.

Note that this routine does not actually enable source blending, nor does it set up the blend operation. It merely sets the following CLT attributes: DBSRCBASEADDR, DBSRCXSTRIDE, DBSRCOFFSET, DBSRCCNTL.

Arguments

g
Pointer to a GState object

Implementation

Library call in libclt.a

See Also

[GS_Create\(\)](#), [GS_GetDestBuffer\(\)](#)



[CLT_InitSnippet](#)

CltEnableTextureSnippet

Global var. used to enable texturing

Description

The CltSnippet **CltEnableTextureSnippet** can be inserted by the user into the current command list buffer when texturing needs to be

enabled. This CltSnippet contains the following commands:

1. Enable the TEXTUREENABLE bit of the TXTADDRCNTL register.

To actually use this CltSnippet, call **CLT_CopySnippetData()**.

Note that if the CltSnippet **CltNoTextureSnippet** is used to disable texturing, using **CltEnableTextureSnippet** will not completely restore state. It will also be necessary for the user's code to set the COLOROUT and ALPHAOUT fields of the TXTTABCNTL register, to either output the texture's colors, or to a blend operation between the primitive's color and the texture's color.

Implementation

Global variable exported in library libclt.a

See Also

CltNoTextureSnippet, **CLT_CopySnippetData()**



CltNoTextureSnippet

CltNoTextureSnippet

Global var. used to disable texturing

Description

The CltSnippet **CltNoTextureSnippet** can be inserted by the user into the current command list buffer when texturing needs to be disabled.

This CltSnippet contains the following commands:

1. Sync the Triangle Engine
2. Disable the TEXTUREENABLE bit of the TXTADDRCTL register
3. Set the TXTABCNTL COLOROUT to PRIMCOLOR, and the ALPHAOUT to PRIMALPHA

To actually use this CltSnippet, call **CLT_CopySnippetData()**.

Implementation

Global variable exported in library libclt.a

See Also

CltEnableTextureSnippet, **CLT_CopySnippetData()**



CltEnableTextureSnippet

Index

A

- aliasing
 - avoiding (with mipmaps) CLT-21
- AllocSignal() CLT-14
- alpha component (of a texel) CLT-28
- alpha value
 - dynamic range of CLT-34
- alpha value, see also alpha component CLT-28

B

- Bi-linear filtering CLT-21
- bi-linear filtering CLT-26

C

- color
 - of 3D objects CLT-17
 - primitive CLT-34
- color component (of a texel) CLT-28
- color components
 - of a texel CLT-32
- color constants CLT-34
- color table CLT-33
- color value, see also color component CLT-28
- command list buffers CLT-10, CLT-11, CLT-13
 - space CLT-13
- command lists CLT-1
- components
 - color (of a texel) CLT-32
 - of a texel CLT-28

- compression
 - data CLT-34
 - of texels CLT-28
- constant registers CLT-28
- constants
 - color CLT-34

D

- data compression CLT-34
- DBALUCNTL CLT-73
- DBAMULTCNTL CLT-69
- DBAMULTCONSTSSB0 CLT-70
- DBAMULTCONSTSSB1 CLT-71
- DBBMULTCNTL CLT-71
- DBBMULTCONSTSSB0 CLT-72
- DBBMULTCONSTSSB1 CLT-72
- DBCONSTIN CLT-69
- DBDESTALPHACNTL CLT-74
- DBDESTALPHACONST CLT-75
- DBDISCARDCONTROL CLT-64
- DBDITHERMATRIX CLT-75
- DBINTCNTL CLT-64
- DblARightJustify attribute CLT-61
- DblBRightJustify attribute CLT-61
- DblEnableAttrs attributes CLT-63
- DblFinalDivide attribute CLT-61
- DblXWinClipMax CLT-63
- DblXWinClipMin CLT-63
- DblYWinClipMax CLT-63
- DblYWinClipMin CLT-63
- DBSRCALPHACNTL CLT-74
- DBSRCBASEADDR CLT-66
- DBSRCCNTL CLT-66

DBSRCOFFSET CLT-67
 DBSRCXSTRIDE CLT-66
 DBSSBDSBCNTL CLT-68
 DBXWINCLIP CLT-65
 DBYWINCLIP CLT-65
 DBZCNTRL CLT-67
 DBZOFFSET CLT-68
 DCNTL CLT-76
 distortion
 and textures CLT-19
 double-buffering CLT-14

E

ESCNTL CLT-77

F

filtering
 bi-linear, see also bi-linear filtering CLT-26
 in texture-mapping CLT-19
 linear, see also linear filtering CLT-24
 mipmap CLT-21
 nearest (point), see also nearest (point) filtering CLT-24
 quasi tri-linear, see also quasi tri-linear filtering CLT-26
 filtering modes
 mipmap CLT-21
 flow control instructions
 INT CLT-9
 JA CLT-9
 JR CLT-9
 PAUSE CLT-9
 SYNC CLT-9
 TXLD CLT-9
 frame buffer bitmaps CLT-13

G

ghost
 rendering CLT-33
 GS CLT-15
 GS_BeginFrame() CLT-14
 GS_Create() CLT-14
 GS_Delete() CLT-15
 GS_FreeBuffers() CLT-15
 GS_GetCurListStart() CLT-14
 gs_ListPtr field CLT-13

GS_Reserve(CLT-13
 GS_Reserve() CLT-13
 gs_SendList CLT-13
 GS_SetList() CLT-14
 GS_SetVidSignal() CLT-14
 GS_WaitIO CLT-14
 GS_WaitIO() CLT-14

I

interfilter CLT-25, CLT-26
 Introduction CLT-2

L

level CLT-20
 level of detail (LOD) CLT-20
 level of detail, see also LOD CLT-20
 Linear filtering CLT-21
 linear filtering CLT-24
 equation for CLT-25

M

magnification filter CLT-25
 mapping
 PIP CLT-32, CLT-33
 texture CLT-19
 minification filter CLT-25
 mipmap
 described CLT-20
 sizes of CLT-20
 mipmap filtering CLT-21
 mipmap filtering modes CLT-21
 mipmap texture filtering CLT-19
 mipmapping CLT-19
 mipmaps CLT-19
 modes
 filtering (for mipmaps) CLT-21
 ModifyGraphicsItem() CLT-15
 multim im parvo CLT-20

N

Nearest (point) filtering CLT-21
 nearest (point) filtering CLT-24, CLT-26

O

object
 textured CLT-19
offset
 PIP CLT-33

P

palette index table, see also PIP table CLT-32
palette lookup table, see also PIP table CLT-32
Pen Index Palette CLT-4
PIP mapping CLT-32, CLT-33
PIP mapping stage CLT-33
PIP offset CLT-33
PIP table CLT-32
PIP-table entry CLT-32
pixel
 perspective correction CLT-6
 texture coordinates CLT-6
 x, y coordinates CLT-6
pixel scaling CLT-61
primitive color CLT-34

Q

Quasi tri-linear filtering CLT-21
quasi tri-linear filtering CLT-26

R

red, green, blue, and alpha components CLT-6
Register Commands CLT-63
registers
 constant CLT-28
rendering a ghost CLT-33

S

scaling
 pixel, see pixel scaling
shift CLT-7
source selection bit, see also SSB CLT-28

space CLT-13
SSB CLT-28, CLT-33
SYNC CLT-4, CLT-9

T

table
 color CLT-33
 PIP, see also PIP table CLT-32
TE, see also Triangle Engine CLT-61
tear-free rendering CLT-13
texel CLT-18
 alpha component CLT-28
 blending CLT-34
 color component CLT-28
 components of CLT-28, CLT-34
 compression of CLT-28
 defined CLT-18
 interpreting color data in CLT-33
 mapping to pixels CLT-19
texture
 address of CLT-18
 coordinate system of CLT-18
 defined CLT-17
texture application blending CLT-36
texture filtering CLT-35
 example of CLT-24
 mipmap CLT-19
texture mapping CLT-19
texture mapping step by step CLT-35
texture RAM, see also TRAM CLT-28
texture-mapping
 example of CLT-33
Textures
 tiled CLT-6
textures
 storing CLT-18
TRAM CLT-33
TRAM (texture RAM) CLT-28
transparency
 of 3D objects CLT-17

Triangle Engine

color calculations performed by CLT-61

deferred mode CLT-4

Destination Blender CLT-2

state registers

shifts and masks CLT-7

Texture Mapper CLT-2

tri-linear filtering

quasi, see also quasi tri-linear filtering CLT-26

U

u coordinate CLT-18

uint32 CLT-34

V

v coordinate CLT-18

vertex header instruction CLT-5

W

WinClipInEnable CLT-63

WinClipOutEnable CLT-63



3DO M2 Link/Dump Programmer's Guide

Version 2.7 – June 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

Preface

About This Document.....	LNK-v
About the 3DO M2 Link/Dump Programmer's Guide	LNK-v
Audience.....	LNK-v
How this Document is Organized	LNK-v
Typographical Conventions	LNK-vi

1

Using Link3do and Dump3do

Introduction.....	LNK-1
About This Chapter	LNK-1
Overview.....	LNK-1
Creating a DLL	LNK-2
Importing From a DLL	LNK-2
Creating and using DLLs	LNK-4
Step 1: Create t2.dll	LNK-4
Step 2: Create t3.dll	LNK-6
Step 3: Create t.elf, Importing From t2.dll and t3.dll	LNK-7

2

Reference Pages

Introduction.....	LNK-11
Link3do	LNK-12
Dump3do	LNK-15

Preface

About This Document

This document provides:

- ◆ A tutorial-style introduction to using dynamic linked libraries
- ◆ References pages for Link3do and dump3do

About the 3DO M2 Link/Dump Programmer's Guide

The 3DO M2 Link/Dump Programmer's Guide describes the linker (Link3do), an MPW tool used to link objects and libraries together in order to create 3DO applications and dynamic link libraries (DLLs). Dump3do, an MPW tool used in conjunction with Link3do, generates a text file of the symbols exported by the DLL file.

Audience

This document is for C programmers preparing titles for the M2 Development System.

How this Document is Organized

This manual is organized as follows:

- ◆ Chapter 1, "Using Link3do and Dump3do," provides a tutorial-style introduction to using dynamic linked libraries.
- ◆ Chapter 2, "Reference Pages," contains references pages for Link3do and Dump3do.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>Scene_GetStatic(scene)</code>
procedure name	<code>Char_TotalTransform()</code>
new term or emphasis	In M2, <i>characters</i> are objects that can be displayed on the screen.
file or folder name	The <i>remote</i> folder, the <i>demo.scr</i> file.

Using Link3do and Dump3do

Introduction

The linker (Link3do) is an MPW tool used to link objects and libraries together in order to create 3DO applications and dynamic link libraries (DLLs). Dump3do is an MPW tool that generates a text file of the symbols exported by the DLL file.

About This Chapter

This chapter discusses the 3DO Link/Dump tools in general terms, then presents a walk-through tutorial that describes how to create and use DLLs.

Topics include:

Topic	Page
Overview	1
Creating and using DLLs	4

Overview

Dynamic linked libraries (DLLs) make it possible for an executable to reference symbols at run-time. DLLs contain code that can be shared among tasks and loaded by the operating system on demand.

The advantages to using DLLs over static linking include:

- ◆ Smaller application size because DLL code is not statically appended to the executable
- ◆ Loading of code that might not be executed can be delayed until the program explicitly requests it. This is called demand loading.

The disadvantage to using DLLs over static linking is that load-time is increased because the loader must load the shared code/data and fix up referenced symbols at load-time/run-time.

Creating a DLL

To use a DLL, you need to create a definitions file that defines the symbols the object will import and/or export. This file lists the symbols a source file needs to reference (import) or export. The linker uses this file to resolve symbols at link time and include information in the binary for use by the loader. The syntax for the definitions file is described in Example 1-2.

The definitions file must state the DLL's library ID and list what symbols defined in the object are to be exported. A unique library ID, or "module ordinal" number, is needed to enable the loader to identify this DLL. You link the object with the definitions file using the `-x` option; the resulting binary is the DLL.

Importing From a DLL

You can specify imports in two ways:

- ◆ You can create a definitions file that explicitly specifies what symbols to import and from what DLLs.
- ◆ You can link with the DLL itself. In this case, the linker simply searches through the files specified on the command line for objects (DLLs) that contain the symbols as exports and automatically imports the symbols it needs from the DLLs.

The advantage of linking through an imports definitions file is that you don't have to list all your DLLs on the link command line. This is particularly convenient when you want to import something from a DLL you don't have. In that case you create a definitions file with the symbols you want to import, including the library number and symbol numbers you want to import.

The library ordinal identifies the unique library ID, while the symbol ordinal identifies the symbol within that library.

Flags (see Example 1-1) can be specified to tell the loader when and how to import the DLL.

Example 1-1 *Flags.*

```
IMPORT_NOW - load the DLL at the time the executable is loaded
              (import at load-time)
REIMPORT_ALLOWED - allow the DLL to be reloaded
IMPORT_ON_DEMAND - wait to load the DLL until a system call is
                  issued from the executable to do so (import at run-time)
IMPORT_FLAG <number> - arbitrary flag number (for future use)
```

Example 1-2 defines the syntax grammar for definitions files. In this example,

- ◆ [x] - x is optional
- ◆ <x> - x is replaced by literal
- ◆ x* - x repeated any number of times
- ◆ x | y - either x or y

Example 1-2 *Syntax grammar for definitions files.*

```
definitions_statement*
definitions_statement->
    !<comment>
export_definitions
import_definitions
import_definitions->
IMPORTS imports [imports_flags]
export_definitions->
MODULE <dll_ordinal> EXPORTS exports
exports->
<export_ordinal>=<symbol_name>
imports->
<symbol_name>=<dll_ordinal>.<export_ordinal>
imports_flag->
flag_name dll*
dll->
<dll_name> | <dll_ordinal>
flag_name->
IMPORT_FLAG <number>
IMPORT_NOW
REIMPORT_ALLOWED
IMPORT_ON_DEMAND
```

Creating and using DLLs

This section describes how to create and use DLLs. An application, *t*, wants to import symbols from two DLLs: *t2.dll* and *t3.dll*. The process involves three basic steps which, for convenience, have been labeled as such:

- ◆ Step 1: Create *t2.dll*
- ◆ Step 2: Create *t3.dll*
- ◆ Step 3: Create *t.elf*, Importing From *t2.dll* and *t3.dll*

Each of the DLLs uses a “definitions” file that defines what that DLL will export. The application *t* also uses a definitions file to show what it wants to import from *t2.dll*, and to specify how to import *t3*.

However, no definitions files are needed if an application is linked with a DLL and the default import flags are sufficient (the default is `IMPORT_NOW`—to load the DLL when the application is loaded).

Note: When using *createm2make*, the linker options below would be automatically generated into the makefile when the option to create a DLL is used.

Assume that we've already created objects *t.o*, *t2.o* and *t3.o* (see compiler options for how to create objects).

Step 1: Create *t2.dll*

Let's look at the link line for *t2.dll*.

```
link3do -r -x t2.def t2.o -Hversion=10 -Hrevision=2 -o t2.dll
```

In this example:

- ◆ `-r` makes it relocatable
- ◆ `-x` says here's the definitions file for this object
- ◆ *t2.o* is the object, containing the symbols to be exported
- ◆ `-Hversion=10` specifies to use version 10 of the application
- ◆ `-Hrevision=2` specifies to use revision 2 of the application
- ◆ *t2.dll* is what we call the DLL

The linker takes the object *t2.o*, reads the definitions, and creates a DLL called *t2.dll*. It stores the library number, and what it exports, in the export section of that file at the ordinal numbers specified in the sample definitions file.

Example 1-3 is the definitions file for this DLL. It contains the export definitions for *t2.c*. Here we:

- ◆ define the library ordinal (or magic number) of this DLL to be 12
- ◆ export the symbol `lproc2` at ordinal 0
- ◆ export the symbol `ldata2` at ordinal 3

- ◆ export the symbol fproc2 at ordinal 2
- ◆ export the symbol fdata at ordinal 4

Example 1-3 File *t2.def*: Export definitions for *t2.c*.

```
MAGIC
12
EXPORTS
!syntax: <ord>=<symbol>
0=lproc2!ordinal 1 is skipped
3=ldata2!order does not matter
2=fproc2
4=fdata
```

Example 1-4 is the file *t2.c* used to create the DLL that exports the symbols *ldata2*, *fdata2*, *lproc2*, and *fproc2*.

Example 1-4 File *t2.c*: Source for *t2.dll*.

```
long ldata2=2;
float fdata2=2.2;

long lproc2(void) {
    return ldata2;
}
float fproc2(void) {
    return fdata2;
};
```

Use the following command to see the symbols exported:

```
dump3do -d t2.dll -o t2.dmp
```

Table 1-1 represents a dump of *t2.dll* after it was linked and built. Note the correspondence of “name” and “index” in this table with the symbols and ordinals respectively in Example 1-3.

Table 1-1 File t2.dmp: Dump of dynamic data for t2.dll.

3DO Exports (libID 12; 5 entries)				
index	secum	offset	symidx	name
0	2	x0	xa	lproc2
1	0	x0	x0	
2	2	xc	xc	fproc2
3	5	x0	xb	ldata2
4	5	x4	xd	fdata2

Step 2: Create t3.dll

```
link3do -r -x t3.def t3.o -Hversion=10 -Hrevision=3 -o t3.dll
```

Example 1-5 contains the export definitions for t3.c.

Example 1-5 File t3.def: Export definitions for t3.c.

```
!t3.def
MAGIC
    13

EXPORTS
    0=vproc3
```

Example 1-6 is the file used to create the DLL t3.dll that exports the symbol vproc3.

Example 1-6 File t3.c: Source for t3.dll.

```
/* t3.c */

long ldata3 = 3;

void vproc3(long l) {
    ldata3 = l;
}
```

Use the following command to get a dump of the symbols exported:

```
dump3do -d t3.dll -o t3.dmp
```

Table 1-2 represents a dump of the Elf file t3.dll. Again note the agreement of symbol and ordinal with that in Example 1-5.

Table 1-2 File t3.dump: Dump of dynamic data for t3.dll.

3DO Exports (libID 13; 1 entries)				
index	secnum	offset	symidx	name
0	2	x0	xa	vproc3

Step 3: Create t.elf, Importing From t2.dll and t3.dll

```
link3do -r -x t.def t3.dll t.o -o t.elf
```

Example 1-7 File t3.def: export definitions for t3.c.

```
!This is a comment

!t.def
!      This file contains import definitions for t.c

IMPORTS
    !syntax: <name>=<lib>.<ord>
    lproc2=12.0 !ordinal 1 is skipped
    ldata2=12.3 !order does not matter
    fproc2=12.2
    !exports from t3 are imported implicitly

IMPORT_NOW
    12      !set flags for how to import the Dlls
           t3.dll!can either use name or module number

REIMPORT_ALLOWED
    12
```

Example 1-8 *File t.c: Source for t.elf.*

```
/* t.c

    This file imports symbols ldata2 , lproc2, and fproc2
    from the DLL t2.dll, and vproc3 from t3.dll.
*/

extern long ldata2;
extern long lproc2(void);
extern float fproc2(void);
extern void vproc3(long);

void main(void) {
    long l;
    float f;
    l = lproc2();
    if (l==ldata2)
        f = fproc2();
    vproc3(l);
}
```

Use the following command to see the symbols that were imported:

```
dump3do -r -t -d t.elf -o t.dmp
```

Table 1-3 *Dump of Elf file t.elf:*

Relocation entries for section ".text" (index2)			
offset	relInfo	symbol	addend
0x10	202:impre124	x00c0000:(lib 12, sym 0)	0x0
0x1a	198:impaddr16ha	x000c0003:(lib 12, sym 3)	0x0
0x1e	196:impaddr16lo	x000c0003:(lib 12, sym 3)	0x0
0x28	202:impre124	x000c0002:(lib 12, sym 2)	0x0
0x30	202:impre124	x000d0000:(lib 13, sym 0)	0x0

Symbol table ".symtab".

index	value	size	bind	type	section	name
0	0x0	0	loc	null	und	
1	0x0	0	loc	sect	abs	t.elf
2	0x0	0	loc	sect	.text	.text
3	0x0	0	loc	sect	.text	.text
4	0x0	0	loc	sect	.debug	.debug
5	0x0	0	loc	sect	.line	.line
6	0x0	0	loc	sect	.strtab	strtab
7	0x0	0	loc	sect	.symtab	.symtab
8	0x0	72	glob	func	.text	main
9	0x0	0	glob	null	und	lproc2
10	0x0	0	glob	null	und	ldata2
11	0x0	0	glob	null	und	fproc2
12	0x0	0	glob	null	und	vproc3

3D0 Imports (2 entries)

index	name (index)	lib_code	lib_ver	lib_rev	flags
0	---(x0)	xc	x0	x0	x3
1	t3.dll (x1)	xd	xa	x3	x1

Reference Pages

Introduction

This chapter contains references pages for the Link3do and Dump3do commands. These commands and a short description of each, include:

Command	Description
Dump3do	Convert libraries or object files to text.
Link3do	Link object files

Link3do

Link object files.

Synopsis

```
link3do [options]...
```

Description

Link3do links object files, dynamically linked libraries (DLLs), archive libraries, and parses import/export definitions files to create elf (symbols) files and DLLs.

Arguments

@argfile	TRead arguments from "argfile."
-	This help.
-B[d=data_base,t=text_base,i=image_base]	Set [data&bss text image] base.
-esymbol	Use "symbol" as entry point.
-llname	Use library lib "lname".a
-m[2 fname]	Generate map file to standard out [fname].
-ofname	Output file (default is a.out).
-r	Generate relocations in file to keep file relative.
i	Incremental link.
-s[s]	Strip unnecessary stuff from file [.comment too].
-Lpath	Add library search path.
-b[secname]=base	Set section base.
-A=alignmen	Set section alignment.
-xdef_file	Use definitions file for resolving imports/ exports.
-v	Verbose.
-g	Generate debug information in file.
-G	Generate debug information to external .sym file.
-k	Keep everything in file.
-n	Generate standard Elf file.
-D	Allow duplicate symbols.
-U	Allow undefined symbols.
-Hfield-val	Set field in the 3do header to "val." Field names: -Hname=<n>: Set the application name. -Hpri=<n>: Set the application priority.

-Hversion=<n>: Set the application version.
 -Hrevision=<n>: Set the application revision.
 -Htype=<n>: Set the application type.
 -Hsubsys=<n>: Set the application subsystem type.
 -Htime <'MM/DD/YY HH:MM:SS' or "now">:
 Set the application time.
 -Hosversion=<n>: Set the application osversion.
 -Hosrevision=<n>: Set the application
 osrevision.
 -Hfreespace=<n>: Set the application freespace.
 -Hmaxusecs=<n>: Set the application maxusecs.
 -Hflags=<n>: Set the application flags.
 -Mmagicno Use "magicno" as the magic number.
 -V Print the version of the linker.

Options set by default are listed in Table 2-1.

Table 2-1 *Default options.*

Option	Description	Value
-A	section alignment	16
-e	entry point	"_start"
-g	generate debug information if any object has a .debug section	
-b	base addresses	0
-o	output filea.out	
-s	strip the symbols from the executable unless generating debug information	

All other options are either 0 or off.

Use the following options for 3DO development:

- ◆ -r create a relocatable executable
- ◆ -lc link with libc.a, and any other needed libraries
- ◆ -Lpath specify the path to the libraries

Always use the -r option when creating a 3DO application. This allows the application to be relocated by the loader.

The -s option causes the linker to create small executables that load quickly. Use this option when creating executables for production.

Return Value

Zero if successful, else failure.

Implementation

Exports.

Here's a sample definitions file.

```
MAGIC          17
EXPORT          !<ord>=<symbol>
                0=dproc
                2=fproc
```

Associated Files

io.h

Caveats

Elf version must match that of OS.

See Also

Dump3do

Dump3do

Convert libraries or object files to text.

Synopsis

```
dump [options] files...
```

Description

Dump elf files, objects, DLLsa, and archive libraries.

Arguments

-o<filename>	Name of output file.
-?	This Help.
-h	File headers (elf header and 3do header).
-p	Program headers.
-s	Section headers.
-t	Symbol table.
-r	Relocation entries.
-d	Dynamic sections (.dynamic, .imp3do, .exp3do).
-a	Hash table contents.
-g	Debug information.
-c	Section contents.
-l	Line information.
-i	Interpret symbolic information.
-b	Binary.
-m	If library, list members.

Return Value

Zero if successful.

See Also

link3do

